Universitat de Lleida
Escola Politècnica Superior
Enginyeria Tècnica en Informàtica de Sistemes

Treball de final de carrera

Generación automática de formularios con asistencia al usuario para la edición de datos en la Web Semántica

Autor: Joan Manel Giménez Méndez
Director: Roberto García González
05 / 2012

Bloque introductorio

Agradecimientos

Quiero presentar mis más sinceros agradecimientos a Roberto García, director del proyecto, por la atención y ayuda prestada en todo momento. Por la implicación mostrada en mi proyecto, y por facilitarme el trabajo proporcionándome las tareas siempre bien definidas y detalladas, así como toda la información que pudiera necesitar para desarrollarlas. También quiero agradecer a la Sala Griho que me hayan proporcionado el material necesario (PC y programas) y el buen trato recibido.

Por otro lado, me gustaría agradecer a mi familia el apoyo ofrecido durante esta etapa de mi vida, así como en cualquier otra.

Resumen

Este proyecto consiste en la adición y la modificación de algunas funcionalidades de una herramienta existente llamada *Rhizomer*. Rhizomer es una aplicación destinada a la publicación de datos semánticos en la Web, por lo tanto, el proyecto está estrechamente relacionado con la Web Semántica.

El proyecto se inicia implementando un formulario dinámico para la localización de recursos para conjuntos de datos en formato RDF. El formulario se genera automáticamente en función de las propiedades específicas de un tipo de datos. Para generar el formulario dinámico de consulta, el usuario debe presionar alguno de los enlaces situados en la página índice de la aplicación, con los distintos tipos de recurso contenidos en el repositorio local de datos.

El formulario ofrece la posibilidad de agregar entradas adicionales, de modo que si el usuario desea especificar el valor para una propiedad que no se encuentra en el formulario generado automáticamente, simplemente tiene que añadir una nueva entrada con la propiedad respectiva. El sistema para añadir propiedades se basa en una entrada de texto con funcionalidad de auto-completado, de forma que cuando el usuario introduce una cadena de texto en la entrada correspondiente, se habilita una lista con las propiedades genéricas (válidas para cualquier tipo de dato) definidas en las ontologías que estrucutran los datos contenidos en el repositorio local, que comiencen de la misma forma que la cadena introducida.

Más adelante, el formulario dinámico de consulta se sustituye por otros sistemas de localización de recursos. No obstante, se adaptan las principales características del formulario dinámico para la generación de formularios de edición de recursos. A partir de entonces, se trabaja sobre los formularios de edición.

Los formularios de edición disponen de la función de auto-completado para los campos de los valores, además de en la entrada para añadir propiedades. Para editar el valor de la propiedad de un recurso, se requiere la selección de uno de los elementos de la lista de resultados que proporciona el auto-completado, es decir, de los recursos contenidos en el repositorio local de datos. En cualquier otro caso, el campo editado se marca como erróneo.

Para ampliar las posibilidades de la herramienta, se implementa un formulario para la definición de descripciones semánticas. De esta forma, cuando el usuario quiere modificar la propiedad de un recurso con un valor no disponible en el repositorio local de datos, tiene la posibilidad de definir la descripción personalmente mediante el formulario generado automáticamente.

El formulario contiene como entradas algunas propiedades genéricas cuyos valores son asignados en función de la propiedad relacionada y de la cadena introducida por el usuario, aunque son editables. Además, también ofrece la opción de añadir más propiedades, y dispone de la función de auto-completado tanto en entradas de propiedades como de valores.

Para terminar, se incluye la posibilidad de realizar las consultas para el auto-completado sobre una base de datos externa. De esta forma, cuando el usuario está editando el valor de una propiedad, si no dispone del recurso deseado en el conjunto de datos local, puede consultar en otro servidor antes de definirlo por su cuenta.

Índice de contenidos

Índice de tablas y figuras

Tablas

Figuras

Introducción

El propósito del proyecto es añadir nuevas funcionalidades a una herramienta existente llamada Rhizomer para ofrecer los medios necesarios para que el usuario sea capaz de corregir posibles errores en un conjunto de datos semánticos, con el fin de mejorar la calidad de los datos con los que se trabaja. Para modificar los datos, se generan formularios dinámicos que ofrecen mecanismos de asistenica para guiar al usuario durante el proceso de edición. Por lo tanto, para situarnos en el contexto de trabajo, se considera necesario incluir una explicación del funcionamiento y la finalidad de la herramienta citada.

Rhizomer es una aplicación de utilidad para la publicación de datos en la Web y su posterior exploración. Los conjuntos de datos con los que trabaja Rhizomer están basados en la Web Semántica y Linked Data, con el fin de dotar a los diferentes tipos de datos de una semántica y una estructura, así como de conectar distintos tipos de datos mediante relaciones, con la finalidad de obtener conjuntos de datos más usables para que el usuario pueda distinguir y entender los distintos tipos de recursos disponibles, las propiedades que contiene cada recurso y las relaciones que pueda haber entre varios recursos (Brunetti et al., 2011).

Esta aplicación, a parte de la visión específica de cada recurso, aporta una visión global del conjunto de datos, de manera que el usuario puede reconocer fácilmente su estructura, y por tanto puede explorar el conjunto de datos de forma práctica y sencilla. Para ello, Rhizomer utiliza unos menús de navegación con las diferentes clases de recursos organizados en función de las ontologías que los clasifican y de su cardinalidad, mostrando en un sitio destacado los tipos con más instancias, y colocando los tipos con pocos recursos asociados en un lugar menos relevante de la barra de navegación. Las clases de recursos que no dispongan de espacio suficiente en la barra de navegación (que serán las que contienen menos instancias) son agrupadas y se muestran a través de un único elemento. Al estar trabajando con datos semánticamente dotados y estructurados por ontologías, es posible automatizar gran parte del proceso de generación y mantenimiento de los menús de navegación. De hecho, éstos se generan dinámicamente cada vez que el conjunto de datos se modifica para mantener siempre actualizados los menús.

Una vez se ha seleccionado una clase o un tipo del menú de navegación, Rhizomer proporciona un sistema de búsqueda exploratoria basado en facetas para facilitar la localización de un recurso específico entre el subconjunto de datos seleccionado. Consiste en una colección de filtros para las principales propiedades de cada tipo de recurso que permiten al usuario realizar consultas para refinar la búsqueda sin tener conocimiento previo de la estructura de datos. Esto significa que el usuario dispondrá de un conjunto de filtros, uno para cada propiedad asociada al tipo seleccionado, donde puede indicar el valor que desee para las distintas propiedades, de forma que se mostrará solamente el subconjunto de recursos que cumpla con las restricciones especificadas en los filtros. De la misma forma que ocurre con los menús de navegación, gracias a los principios de la Web Semántica, gran parte del proceso de generación y mantenimiento de las facetas es automático. Se generan a partir de consultas SPARQL realizadas a cada tipo definido en las ontologías, donde se extraen todas las propiedades de cada tipo, así como los distintos valores para cada propiedad y la cardinalidad de cada valor. Una vez obtenidas las facetas, se almacenan en una estructura de datos y cada vez que el conjunto de datos sufre algún cambio son actualizadas. Las facetas se generan de forma genérica y escalable, de modo que no

dependen de la estructura de un conjunto de datos en concreto, sino que pueden procesar datos provenientes de distintos orígenes.

Una vez seleccionado uno de los recursos a través de las facetas, la atención se centra en la visión concreta del recurso: una lista con las principales propiedades del recurso. Rhizomer incorpora un conjunto de servicios que se asocian dinámicamente a los recursos en función de su tipo para mejorar la presentación de los datos aumentando la interacción y consiguiendo una vista más sugerente. Por ejemplo, posibles servicios serían calendarios para mostrar fechas (se asociarían a recursos que tengan alguna propiedad con formato de fecha) o mapas para mostrar puntos geográficos (se asociarían a recursos que tengan alguna propiedad representando un punto geográfico o bien sus coordenadas).

Motivación

La motivación general para este proyecto es conseguir una publicación de datos semánticos más sostenibles. Se trata de ofrecer herramientas para que el usuario, que en la mayoría de ocasiones es quien se percata de los errores que pueda haber en los datos mostrados, sea capaz de modificar los datos con el fin de reparar los errores y mejorar la consistencia de los mismos. Se pretende dar la posibilidad al usuario de editar los recursos mediante formularios dinámicos generados automáticamente en función del tipo de recurso y de sus propiedades. El problema que podemos encontrar aquí es que la mayoría de usuarios tendrán un conocimiento limitado del conjunto de datos y de la Web Semántica, lo que implica que lo más probable es que cuando un usuario modifique la propiedad de un recurso, lo haga de forma incorrecta, y por lo tanto provoque inconsistencia en el sistema. Este es un importante apartado a tener en cuenta, ya que para mantener el conjunto de datos consistente es necesario comprobar que todos los datos introducidos o modificados deben respetar las restricciones impuestas por los esquemas y ontologías subyacentes.

Para mantener la consistencia del sistema, se le proporcionará al usuario asistencia durante el proceso de edición. El asistente consiste en un mecanismo de auto-completado para las entradas de los formularios, que suministra al usuario los posibles valores para el campo en edición en función de las restricciones impuestas por los esquemas y ontologías utilizadas, fomentando la utilización de valores existentes o de conjuntos de datos referencia. De esta forma, el usuario es capaz de corregir fácilmente posibles errores en el conjunto de datos a través del navegador, minimizando al mismo tiempo la posibilidad de que los datos introducidos por el usuario impliquen alguna inconsistencia.

Objetivos

Este proyecto se emprende con el fin de añadir nuevas funcionalidades a una aplicación existente para ampliar los servicios ofrecidos por la herramienta. El propósito general es conseguir una publicación de datos semánticos más sostenibles para mejorar la calidad de los conjuntos de datos con los que trabaja. Para conseguirlo, se han definido varias metas a lo largo del proyecto, de tal forma que no se dispone de la totalidad de los objetivos desde el inicio, sino que a medida que se va avanzando en el desarrollo del proyecto, se van definiendo en función de las carencias y necesidades que surgen después de comprobar los resultados obtenidos en la última aportación. No obstante, se exponen los principales objetivos definidos durante el desarrollo del proyecto:

- Implementación de un formulario dinámico para la localización de recursos en conjuntos de datos en formato RDF.
- Implementación de mecanismos de asistencia basados en un sistema de entradas de texto con función de autocompletar.
- Actualización del código referente a 'Yahoo! User Interface Library' contenido en el proyecto a la nueva versión (YUI 3) para reparar posibles bugs y disponer de las mejoras realizadas.
- Implementación de formularios dinámicos para la definición de descripciones semánticas para que el usuario pueda introducir en los campos de los formularios de edición recursos no contenidos en el repositorio local de datos.
- Modificación del objeto AutoComplete para que realice las consultas correspondientes sobre fuentes de datos remotas y no se tenga que almacenar el conjunto de datos con el que trabaja en la página actual.
- Ofrecer la posibilidad al usuario de realizar las consultas que lanza el mecanismo de auto-completado durante el proceso de edición de un recurso sobre el conjunto de datos definido, en un repositorio de datos externo, almacenado en un dominio distinto al que pertenece la aplicación.

Temporalización / Presupuesto

Dado que, por la tipología del proyecto realizado, no se han marcado desde el principio unos objetivos finales, sino que se han ido definiendo las propuestas sobre la marcha, es difícil realizar una planificación detallada. Sin disponer de los objetivos finales, la planificación se tiene que ir definiendo a medida que se especifican los objetivos, por lo que la desviación entre la planificación prevista y el desarrollo real es mínima, de manera que pierde un poco el sentido realizar una comparativa detallada. Por lo tanto, en su lugar se expondrá la temporalización real aproximada para las distintas fases del proyecto organizadas en iteraciones.

Iteración	Duración
Trabajo Previo	8 semanas
1 Formulario genérico de consulta	½ semana
2 Intérprete SPARQL para el formulario Web	2 semanas
3 Formulario específico de consulta	2 semanas
4 Formulario dinámico de consulta	1 semana
5 AutoComplete	1 semana
6 Diálogo de confirmación	½ semana
7 Formulario para la definición de descripciones semánticas	1 semana
8 Confirmar valor mediante 'enter'	½ semana
9 De YUI 2 a YUI 3	4 semanas
10 Trabajar con datos en JSON	½ semana
11 AutoComplete en tiempo real	2 semanas
12 Nueva descripción semántica como primer resultado de AC	1 semana
13 Consultas de AutoComplete en fuentes de datos externas	2 semanas
14 Bases de datos del AutoComplete intercambiables	½ semana
15 Consultas a DBpedia con restricción de tipo	2 semanas
16 Refactoring	½ semana
TOTAL	**29 semanas**

Tabla 1: Temporalización

Las diferentes iteraciones se han desarrollado en varias etapas, en cada una de las cuales se ha aplicado distinta intensidad. Es decir, se han dedicado más o menos horas diarias a desarrollar el proyecto en cada etapa por motivos extra-académicos, dependiendo de la circunstancia personal en cada etapa o momento.

Por lo tanto, para calcular un presupuesto aproximado, se establece una jornada media de trabajo de 20 horas semanales.

Por otro lado, dado que una parte del trabajo realizado ha consistido en la formación de nuevas tecnologías, se establece un coste aproximado para un trabajo en prácticas de 10€/hora.

Por lo tanto, el coste total estimado de la elaboración del proyecto desarrollado es de 29 semanas x 20h/sem x 10€/h = 5800€.

Estructura del resto del documento

A continuación encontramos el bloque de ‘Estado del arte’, el cual está dividido en tres secciones. En la primera, ‘Tecnologías relacionadas’, se desarrolla una amplia explicación de las principales tecnologías empleadas para llevar a cabo el trabajo. Dentro de esta sección se habla de la Web Semántica, la principal tecnología en la que se basa la herramienta Rhizomer, y por lo tanto el proyecto. Este apartado incluye una breve explicación de algunos lenguajes estrechamente ligados a la Web Semántica, como RDF, OWL o SPARQL. En esta sección también hay un apartado para JavaScript, que es el lenguaje de programación por excelencia de la parte del cliente en aplicaciones web, y como consecuencia, el principal lenguaje utilizado para la elaboración y el desarrollo del proyecto. También hay un apartado para AJAX, tecnología para la programación de aplicaciones interactivas; y el último corresponde a *Yahoo! User Interface Library*, donde se explican las principales librerías que se han utilizado de YUI Library a lo largo del proyecto para añadir algunos widgets y utilidades.

La sección de ‘Estudios realizados’ consiste en un listado de las tecnologías que han necesitado un estudio previo al inicio del proyecto para entender de qué tratan y qué se proponen, y por supuesto para aprender a utilizarlas.

En la sección de ‘Otros trabajos similares’, se exponen tres aplicaciones parecidas o con algunas funcionalidades semejantes a la aplicación en la que se ha estado trabajando durante el desarrollo del proyecto, además de una breve comparación del funcionamiento entre dichas aplicaciones y el resultado del trabajo realizado sobre Rhizomer.

En el bloque de ‘Desarrollo’ es donde encontramos el contenido del trabajo realizado durante el desarrollo del proyecto. Consta de dos secciones: ‘Gestión del proyecto’ y ‘Desarrollo’. En ‘Gestión del proyecto’ se expone brevemente el método usado para el desarrollo del trabajo y se definen las diferentes etapas y su evolución en el tiempo. En ‘Desarrollo’, se explica detenidamente el trabajo realizado a lo largo del proyecto. Este apartado está dividido en varias iteraciones, cada una de las cuales corresponde a un funcionalidad o utilidad implementada/modificada secuencialmente ordenadas.

El bloque de 'Finalización' contiene una sección de ‘Conclusiones’, donde se exponen, por un lado, las conclusiones derivadas del trabajo realizado, es decir, si se han cumplido los objetivos propuestos; y por otro lado, conclusiones a nivel personal. Para acabar, la última sección este bloque, ‘Trabajo Futuro’, habla de aspectos a mejorar o posibles funcionalidades a añadir a la aplicación, que quedan pendientes de realizar en un futuro próximo.

Bloque de estado del arte

Tecnologías relacionadas

Web Semántica

Introducción

La Web se ha convertido en una herramienta imprescindible en el día a día de nuestra sociedad. En un principio, era un simple escaparate de información, donde se podía consultar cualquier tipo de información procedente de cualquier lugar del mundo, pero nada más. Dado al éxito obtenido, la Web ha ido evolucionando de forma continua, hasta tal punto que hoy en día contiene millones de aplicaciones interactivas y ofrece multitud de servicios, que permiten realizar actividades de todo tipo de una forma tan cómoda, sencilla y económica, inimaginable hace pocos años. Por ello, no sólo se ha convertido en un medio de comunicación y difusión tan importante como el teléfono o la televisión, sino que cada vez es más relevante en más sectores de nuestra sociedad: servicios, ocio, negocios...

En los últimos años, la cantidad de recursos disponibles en la Web ha aumentado exponencialmente. El enorme tamaño que ha alcanzado la Web es una de las claves de su éxito, pero también supone un aumento considerable del trabajo o del tiempo que implica llevar a cabo ciertas tareas. Y tal y como están codificados actualmente los contenidos Web, resulta excesivamente complicado y costoso desarrollar y mantener programas que realicen estas tareas en nuestro lugar.

El gran crecimiento de la Web implica que las tecnologías que la hacen posible evolucionen para adaptarse a las circunstancias en cada momento para alcanzar los objetivos propuestos: lograr una Web más amplia, ofrecer más servicios y posibilidades, automatizar procesos y tareas engorrosas…

Para solucionar los problemas comentados, a finales de los 90 surge el concepto de *Web Semántica*, con el propósito de lograr que las máquinas entiendan la información que contiene la Web, y por lo tanto, sean capaces de interpretarla y utilizarla con el fin de automatizar algunas operaciones que hoy en día los usuarios efectúan, liberándolos así de parte del trabajo. Para ello, la Web Semántica propone introducir descripciones explícitas sobre el significado de los recursos, de tal forma que las propias máquinas tengan el conocimiento necesario para realizar las tareas más costosas y rutinarias que actualmente llevan a cabo los usuarios que navegan por la red (Castells, 2003). En definitiva, la Web Semántica pretende desarrollar una Web más unida y organizada, donde sea más fácil localizar, integrar y manipular información en la red.

Un ejemplo claro de las carencias de la Web actual son las limitaciones de los buscadores. Cuando realizamos una búsqueda, los resultados serán multitud de enlaces a páginas que contengan la cadena de caracteres introducida, pero sin interpretar el significado de la palabra (una misma palabra puede tener varios significados o diversas connotaciones dependiendo del ámbito o contexto en la que se esté utilizando) ni la relación entre palabras (no es lo mismo un retrato *de X*, que un retrato *pintado por X*). Por este motivo, si necesitamos buscar algo muy concreto o ambiguo, nos vemos en la obligación de realizar varias consultas y leer varios documentos hasta llegar a la respuesta deseada.

Todo esto evidencia la falta de expresividad en la representación de datos en la que se basa la Web actual. Los contenidos Web están presentados en formatos e interfaces

comprensibles por personas, pero no por máquinas, al menos sin un coste excesivo de recursos (por ejemplo, utilizando técnicas como el screen scraping o el procesado de lenguaje natural). Un programa puede transportar, generar, modificar cierta información interactuando con el usuario, pero la máquina no sabe qué significa, por lo que no es capaz de actuar independientemente, sin las directrices del usuario.

Aquí es donde entra la Web Semántica, que pretende superar las limitaciones de la Web actual introduciendo descripciones del significado, la estructura interna y la estructura global de los contenidos y servicios de manera que las máquinas puedan entender la información que están manejando y sean capaces de manipularla correctamente para ejecutar tareas que hasta ahora solo las personas realizaban.

Para llevar a cabo todos estos objetivos, se han creado una gran diversidad de nuevas tecnologías, como lenguajes para la representación de metadatos y ontologías, parsers, lenguajes de consulta, entornos de desarrollo…

El lenguaje RDF

El lenguaje RDF (Resource Description Framework) es un lenguaje de propósito general para la representación de información en la Web mediante descripciones explícitas del significado de los recursos a través de la definición de ontologías y metadatos (González, 2007).

Actualmente, RDF es el estándar más extendido en la comunidad de la Web Semántica. La finalidad de RDF es definir un mecanismo para describir recursos que sea válido para cualquier dominio, para evitar la necesidad de definir un mecanismo concreto para cada dominio diferente.

XML proporciona la sintaxis necesaria para representar datos y estructuras por separado de la presentación en HTML, pero carece de las semánticas necesarias para dotarlos de significado. RDF permite especificar la semántica para bases de datos XML, resolviendo así la carencia de significado mencionada. XML es una sintaxis posible para RDF, pero no la única, ya que RDF puede trabajar sobre diferentes formas de representación.

El modelo RDF consiste en representar las declaraciones de los recursos mediante tripletas del tipo ‘Sujeto-Predicado-Objeto’, donde el sujeto y el objeto son nodos, y el predicado es un arco que los conecta. El sujeto corresponde siempre a un recurso. Si el predicado es una propiedad, el objeto es el valor de la propiedad de dicho recurso. En cambio, si el predicado es una relación, el objeto equivale a otro recurso con el que se establece la relación con el sujeto.

RDF se puede combinar con otras herramientas para ampliar o mejorar su cometido. Por ejemplo, RDF Schema permite definir jerarquías de clases de datos, y OWL permite la definición de ontologías.

Para cumplir el objetivo de desarrollar la Web Semántica, también es necesario un lenguaje de consulta y un protocolo de recuperación estándar. EL W3C ha desarrollado el lenguaje SPARQL (Simple Protocol and RDF Query Languaje) para cubrir estas necesidades, y ya se ha convertido en el lenguaje estándar de consulta para bases de datos en RDF.

SPARQL - Simple Protocol and RDF Query Language

SPARQL es el lenguaje de consulta estándar para bases de datos en RDF, aunque puede expresar consultas para diversas fuentes de datos, sin tener en cuenta la tecnología de la base de datos o el formato utilizado para almacenar los datos (W3C, 2008a).

SPARQL permite escribir consultas en forma de tripletas, conjunciones y disyunciones (AND y OR lógicos), lo que hace bastante sencillo formular consultas para RDF debido a la similitud con el lenguaje en la forma de expresar sentencias. El resultado de una consulta SPARQL puede ser un conjunto de valores o un grafo RDF.

SPARQL proporciona medios para la óptima recuperación y organización de la información. Está separado en tres especificaciones diferenciadas: un lenguaje de consulta (W3C, 2008a), un formato para las respuestas (W3C, 2008b), y un protocolo para el transporte de consultas y respuestas (W3C, 2008c, citado por W3C, 2008a).

Ontologías y OWL

Una de las dificultades que se presentan durante el desarrollo de la Web Semántica es la de llegar a un acuerdo para la representación de la realidad. Para salvar este obstáculo, la Web Semántica propone la representación del conocimiento mediante ontologías, un concepto proveniente de la Inteligencia Artificial.

Tal y como define Castells (2003), una ontología es una jerarquía de conceptos con atributos y relaciones, que define una terminología consensuada para definir redes de unidades de información interrelacionadas.

Esto significa que las ontologías permiten representar cualquier objeto o concepto a través de tipos o clases. Los diferentes tipos de datos están relacionados entre ellos, determinando una estructura lógica de tal forma que facilita la gestión de grandes cantidades de datos.

Cabe destacar la importancia de acordar ontologías comunes y universales para poder gestionar de forma independiente la información que contienen con garantías de concordancia.

La Web Semántica no sólo propone dotar de significado a la información y los recursos, también pretende automatizar algunas partes de los servicios Web, como su descrubimiento, ejecución, o la comunicación entre distintos servicios. Para conseguir este propósito, la Web Semántica plantea la definición de ontologías de funcionalidad y de procedimientos.

Las entidades de una ontología son lógicas, establecen relaciones lógicas y soportan operaciones lógicas. Las ontologías se pueden basar en diferentes tipos de lógica, de modo que dependiendo de la lógica utilizada, se puede obtener distinto nivel de expresión, de eficiencia y un sistema decidible o no decidible (del Teso, 2007).

En lógica, la decidibilidad consiste en la existencia de un método efectivo para determinar si un objeto es miembro de un conjunto de fórmulas. Un sistema lógico es decidible si todas las fórmulas posibles del sistema son decidibles. Entonces, un problema es decidible si existe un algoritmo capaz de asegurar si tiene o no solución. Aplicando este concepto sobre un lenguaje, resulta que es decidible si existe un

algoritmo que para cada sentencia posible sea capaz de decidir si pertenece a dicho lenguaje.

La lógica más habitual para estructurar ontologías es DL (Description Logic), que es la lógica más expresiva de entre las que garantizan la computabilidad.

El lenguaje más extendido para definir ontologías es OWL. OWL se suele formular en RDF, por lo que se considera una extensión de éste. OWL dispone de toda la capacidad expresiva de RDF Schema, aumentando sus posibilidades al permitir el uso de expresiones lógicas.

Dentro de OWL se pueden distinguir tres niveles, diferenciados por la expresividad y eficiencia, dependiendo éstos de los constructores y axiomas de que disponga la lógica con la que trabajan: OWL-Full, OWL-DL, OWL-Lite. Los tres niveles son válidos, dependiendo claro de las necesidades de cada modelo. OWL-Full es el más expresivo, pero no es decidible. OWL-Lite es el más eficiente, pero pierde mucha expresividad. OWL-DL es el más habitual de los tres por disponer de una gran expresividad y ser a la vez decidible.

OWL-DL está basado en predicados lógicos de primer orden y permite aprovechar todo el poder expresivo de OWL exceptuando algunas limitaciones en la restricción de clases (Pollock, 2009). OWL-DL se basa en la asunción de mundo abierto (OWA - Open World Assumption). La asunción de mundo cerrado (CWA - Closed World Assumption) significa que cualquier declaración que no se ha demostrado que sea cierta es falsa. OWA es todo lo contrario, ninguna declaración es falsa hasta que no se ha probado por completo que es falsa. Esto significa que el sistema asume que la información de la que dispone no está completa, y en caso de no encontrar una respuesta perfecta, no implica que no exista.

DL es una lógica monótona, y por lo tanto, también lo es OWL-DL, que quiere decir que el hecho de añadir nuevas declaraciones a nuestra base de datos nunca desmentirá una conclusión previa.

OWL tiene tres elementos básicos: instancias, propiedades y clases. Las propiedades pueden ser de dos tipos: propiedades de tipo o atributos, y propiedades de objeto o relaciones. Los atributos ayudan a describir instancias mediante valores literales asociados a cada instancia: título, fecha, altura… Las relaciones establecen conexiones entre instancias de dos o más clases.

Las clases pueden limitar o condicionar su alcance a través de restricciones. Las restricciones se especifican mediante operadores lógicos, e indican cómo pueden relacionarse las instancias de las clases asociadas a la restricción.

Por todo esto y mucho más, se puede decir que OWL es más completo que otras formas de representar la realidad como pueden ser las bases de datos relacionales o los sistemas orientados a objetos, destacando en términos generales la precisión, el dinamismo y la expresividad que lo hacen característico.

Retos de la Web Semántica

La enorme cantidad de recursos que contiene la Web hace que la transición de la Web actual a la Web Semántica suponga un problema debido al alto coste que conlleva. Tal y como comenta Castells (2003), las estrategias más viables para llevar a cabo semejante trabajo consisten en combinar una pequeña parte del trabajo manual con la

automatización del resto del proceso, con técnicas como la extracción de metadatos a partir de texto y recursos multimedia, o el mapeo de la estructura de bases de datos a ontologías.

Otra dificultad que se presenta para elaborar la Web Semántica es la adopción de ontologías comunes, ya que cada parte del sistema conlleva peculiaridades necesarias: la representación del mundo depende en gran medida de la perspectiva desde donde se mira, del ámbito o área desde donde se esté trabajando. La solución a esto es la de generar ontologías genéricas que se adapten a cualquier campo, y por medio de extensión o especialización de éstas, crear ontologías adecuadas a cada área, estableciendo formas de compatibilidad entre las distintas ontologías.

La cantidad y calidad de ventajas que presenta la Web Semántica sobre la Web actual o Web 2.0 hace que se haya convertido en un área de investigación importante en los centros de investigación de todo el mundo, así como las principales agencias de financiación pública inviertan grandes presupuestos en proyectos de investigación y desarrollo para la Web Semántica. Cabe destacar el importante papel que el W3C ha ejercido en el proyecto de la Web Semántica, con la creación de grandes grupos de trabajo y de la mayoría de lenguajes y tecnologías estandarizados específicos para la Web Semántica.

JavaScript

Introducción

A principios de los años 90 empiezan a desarrollarse las primeras aplicaciones Web, y con ellas surge la obligación de empezar a incorporar formularios cada vez más complejos. Teniendo en cuenta la velocidad de navegación en la red de la época (un módem que pudiera adquirir un ciudadano corriente trabajaba a una velocidad máxima de 28.8 kbps), muchas tareas requieren demasiado tiempo y empiezan a resultar un tanto tediosas.

Por ejemplo, para enviar un formulario con un supuesto error, se enviaba el formulario al servidor, éste efectuaba las comprobaciones necesarias y posteriormente volvía a enviar el formulario al cliente indicándole los campos erróneos. El resultado de tanta transferencia de datos a la ínfima velocidad a la que viajaban por entonces es un consumo excesivo de tiempo para efectuar una acción de lo más simple. Por motivos como este nace la necesidad de crear un lenguaje de programación que se ejecute directamente en el navegador del usuario.

JavaScript nace como solución a la falta de dinamismo en la Web. Trabajando con HTML solo se pueden crear documentos estáticos de manera que tenemos la Web repleta de información con la que no podemos interactuar.

JavaScript es un lenguaje tipo script o interpretado, lo que significa que un programa JavaScript no requiere ningún tipo de compilación previa a su ejecución, o lo que es lo mismo, cualquier navegador lo puede interpretar y por lo tanto ejecutar.

Es un lenguaje basado en objetos, diseñado para el desarrollo de aplicaciones cliente-servidor. Los programas JavaScript se insertan en documentos HTML y se encargan de realizar acciones y peticiones en el cliente, como comprobar los campos rellenados de un formulario antes de enviarlo al servidor, por ejemplo.

Características del lenguaje

En esta sección se analiza JavaScript (Crockford, 2008), un lenguaje construido sobre buenas ideas y conceptos, pero también con algunas partes discutibles. A modo de resumen, algunos de los buenos conceptos sobre los que podemos trabajar son las funciones, el tipado débil, los objetos dinámicos y una notación literal de objetos muy expresiva. En el otro lado tenemos un modelo de programación basado en las variables globales.

La mejor parte del lenguaje es su implementación de funciones. Las funciones son la unidad modular fundamental de JavaScript y sirven para la reutilización de código, para ocultar información y para la composición y estructuración de información.

Las funciones en JavaScript son objetos. Un objeto es una colección de parejas nombre/valor con un enlace oculto a un objeto prototipo. Cada función contiene dos propiedades ocultas: el contexto de la función y el código que implementa el comportamiento de la función. Cada objeto función también contiene una propiedad de prototipo. Su valor es un objeto con una propiedad constructora, y sirve para que otros objetos hereden de él. Todas las funciones disponen de una propiedad prototipo, aunque sólo será útil para funciones constructoras, pero JavaScript no sabe si una función va a ser constructora en el momento de su definición.

Una función puede ver todo lo que contiene (Lexical Scoping), incluso las variables declaradas dentro de algún bloque (conjunto de declaraciones encerrado entre llaves {}), a diferencia de la mayoría de lenguajes de programación. En cambio, las variables declaradas dentro de una función no son visibles desde fuera de ésta.

Debido a que las funciones son objetos, se pueden usar como cualquier otro valor. Las funciones se pueden almacenar en variables, objetos y vectores. Se pueden pasar como argumentos de otras funciones, y pueden ser devueltas de otras funciones. También pueden tener métodos.

Las funciones se pueden definir dentro de otras funciones. No solo tienen acceso a sus parámetros y variables, sino también a los parámetros y variables de la función en la que está anidada, ya que puede acceder a esta información a través de su propiedad oculta de contexto. A esto se le llama clausura (Closure) y sirve, entre otras cosas, para ocultar información.

Por ejemplo, queremos crear un objeto con atributos privados. En lugar de declararlo con una literal, lo hacemos llamando a una función que devuelva la literal de objeto. No estamos asignando una función a una variable, sino el resultado de invocar la función. Lo que declaramos en una función no es visible desde fuera, así que las variables que declaremos serán atributos privados del objeto. En el bloque de la declaración *return* definimos un objeto con uno o varios métodos, los cuales sí tendrán acceso a los “atributos privados” a través de su propiedad oculta de contexto (clausura). De esta forma, se obtiene un objeto con métodos públicos, capaces de acceder a los atributos y métodos privados, definidos en el cuerpo de la función.

Podemos utilizar las funciones y la clausura para crear módulos. Un módulo es una función o un objeto que presenta una interfaz pero oculta su estado e implementación. El patrón general de un módulo es una función que define variables y funciones privadas; crea funciones privilegiadas que, mediante la clausura, tienen acceso a las variables y funciones privadas; y devuelve las funciones privilegiadas, o bien las almacena en un lugar accesible. Utilizando patrones de módulo podemos eliminar el uso de las variables globales.

Además, a una función JavaScript le podemos pasar menos argumentos que los parámetros que tiene definidos la función, cada uno de los cuales (los que no estén especificados) será sustituido por el valor *undefined*. Le podemos pasar incluso más argumentos que los que admite en su definición, los cuales podemos consultar en un parámetro suplementario de las funciones accesible después de su invocación, que contiene un vector con todos los argumentos pasados a la función, incluyendo los suplementarios que no son asignados a ningún parámetro.

JavaScript reconoce seis tipos de valores diferentes: numéricos, lógicos, objetos, cadenas, nulos e indefinidos. A diferencia de otros lenguajes, JavaScript emplea un tipado débil, lo que significa que una variable puede cambiar de tipo durante su ciclo de vida. Por ejemplo, se puede declarar una variable que ahora sea un entero y más adelante una cadena. Además, en JavaScript no es necesario especificar el tipo de dato en la declaración de una variable, sino que el intérprete asignará el tipo apropiado en cada momento.

La mayoría de lenguajes exigen un tipado fuerte. La teoría es que un tipado fuerte permite al compilador detectar una amplia clase de errores en tiempo de compilación. JavaScript es un lenguaje débilmente tipado, así que los compiladores de JavaScript no son capaces de detectar errores de tipo. No obstante, un tipado fuerte tampoco elimina la necesidad de evaluar detenidamente el código, y los errores que puede detectar respecto a un tipado débil no son relevantes. En cambio, resulta muy práctico y mucho más simple al no tener que formar complejas jerarquías de clases, o no tener que preocuparse del sistema de tipos.

JavaScript tiene una notación literal de objetos muy potente. Los objetos pueden ser creados simplemente enumerando sus componentes. Este tipo de notación fue la inspiración para JSON, un conocido formato de intercambio de datos.

Aunque hay infinidad de objetos predefinidos con los que podemos trabajar en JavaScript, destacaremos los más comunes, cada uno de los cuales posee numerosos atributos y métodos para facilitar y aumentar las posibilidades de su uso:

- Objetos de lenguaje: string, array, math, boolean, date, number, function...
- Objetos del navegador: window, frame, location, history, navigator, document, link, anchor, image, form, text, textarea, button, checkbox, select, hidden…

Otra diferencia con otros lenguajes es que un vector no tiene que tener una longitud definida, y si la tiene se puede cambiar. Se puede insertar un elemento en cualquier posición sin depender de la longitud del vector. Es decir, con otros lenguajes, si tienes un vector de longitud 10 e insertas un elemento en la posición 20, se sale de rango. En JavaScript, inserta el elemento en la posición indicada y aumenta su longitud.

Una de las características polémica de JavaScript es la herencia prototípica. JavaScript tiene un sistema de objetos libre de clases, donde los objetos heredan propiedades directamente de otros objetos. Es una característica muy potente, aunque contraste con los lenguajes clásicos y pueda resultar un tanto compleja o confusa para desarrolladores acostumbrados a lenguajes basados en el concepto clase. Por esta razón, JavaScript también proporciona herramientas para crear sistemas de clases (Herencia Pseudo-Clásica).

Para ello, se añade un nivel innecesario para que los objetos sean producidos por funciones constructoras. Los objetos construidos con este método se definen con el prefijo *new*. Tal y como describe Stefanov (2010), todas las funciones JavaScript tienen una propiedad prototipo, el valor de la cual es un objeto prototipo. En el objeto

prototipo se almacenan todos los rasgos o propiedades que serán heredadas por los nuevos objetos creados mediante su función constructora. A todas las funciones se les asigna un objeto prototipo, ya que el lenguaje no proporciona forma alguna de determinar si una función va a ser constructora.

Podemos crear funciones constructoras que actúen aparentemente como clases, pero no se comportan como tales, ya que carecen por completo de privacidad: todas las propiedades son públicas. Además no tienen acceso a los métodos de las clases superiores. La herencia pseudo-clásica puede proporcionar cierta comodidad a programadores no acostumbrados a JavaScript, pero esta notación basada en sistemas de herencia clásicos puede inducir a los programadores a componer extensas jerarquías que sean innecesariamente profundas y complejas.

Muchas veces, la complejidad de las jerarquías de clases está motivada por las limitaciones de un tipado fuerte y estático en tiempo de compilación. JavaScript no comparte este tipo de limitaciones, por lo que resulta mucho más simple y práctico emplear la herencia prototípica, en la que los objetos heredan propiedades directamente de otros objetos, pudiendo añadir a cada objeto nuevos atributos y métodos. Pero este sistema también presenta un defecto: la falta de privacidad. Para solucionar este problema, JavaScript proporciona la herencia funcional, que consiste en crear objetos a través de funciones que oculten cierta información (atributos y/o métodos) mediante módulos, método explicado anteriormente.

De esta forma se pueden conseguir funciones constructoras que generen objetos con propiedades privadas.

La sintaxis de JavaScript proviene de C. En C y otros lenguajes similares, las variables declaradas en un bloque no son visibles desde fuera del bloque. JavaScript emplea la misma sintaxis, pero no el mismo funcionamiento, ya que una variable encerrada en un bloque es visible desde cualquier punto de la función que contiene al bloque. Es una diferencia a tener en cuenta con la mayoría de lenguajes.

La peor de las características de JavaScript es su dependencia en las variables globales. En JavaScript es fácil crear variables globales que contengan todos los recursos de nuestra aplicación. Las variables globales debilitan la fiabilidad de los programas y deberían ser evitadas. Afortunadamente, JavaScript proporciona las herramientas necesarias para salvar este problema.

Una variable global es una variable visible desde cualquier alcance, desde cualquier parte del programa. Pueden ser apropiadas para pequeños programas, pero resultan difíciles de manejar a medida que crece el programa porque una variable global puede modificarse desde cualquier punto del programa y en cualquier instante, lo que puede complicar gravemente su funcionamiento.

Muchos lenguajes tienen variables globales, el problema con JavaScript es que requiere de ellas, a diferencia de otros lenguajes que simplemente permiten utilizarlas. En JavaScript, todas las unidades de compilación se cargan en un espacio común: el Objeto Global.

Hay tres formas de definir variables globales. Declarándola fuera de cualquier función con la declaración *var*. Otra forma es añadiendo una propiedad al objeto global, que en los navegadores web es el objeto *window*. Esta propiedad será una variable global. También se puede inicializar una variable sin haberla declarado, lo que la convierte en variable global.

Hay que prestar especial atención a esta tercera forma ya que olvidarse de declarar una variable es un error muy habitual y es difícil de encontrar ya que el compilador hará caso omiso al estar permitido no declararlas; y podemos tener una variable global donde no la deseamos que seguramente cambiará el funcionamiento del programa.

Una forma de minimizar el uso de las variables globales es creando una única variable global para toda la aplicación. De esta forma se reduce drásticamente la posibilidad de malas interacciones con otras aplicaciones, widgets o librerías. Otra forma efectiva de reducir el efecto de las variables globales es ocultar información (variables, métodos) mediante la clausura (Closure), método mencionado anteriormente.

Tal y como dice Eguíluz (2009), JavaScript fue diseñado para ejecutarse en entornos muy limitados, para intentar garantizar la confianza de los usuarios en la ejecución de los scripts. Por este motivo, los scripts de JavaScript no pueden comunicarse con otros dominios, ni pueden acceder a los archivos del ordenador del usuario, ni pueden leer ni mucho menos modificar las preferencias del navegador.

Afortunadamente, existen varias alternativas para solucionar los problemas mencionados. La más común consiste en firmar digitalmente el script y solicitar permiso al usuario para efectuar las acciones pertinentes.

Conclusiones

JavaScript es un lenguaje de contrastes. Es un lenguaje con ciertas partes problemáticas que pueden inducir a cometer errores, confusiones o ambigüedades a la hora de escribirlo. Pero la Web se ha convertido en una importante plataforma para el desarrollo de aplicaciones, y habiendo fracasado otros lenguajes tan importantes como Java en este entorno, JavaScript es el único lenguaje que se encuentra en todos los navegadores. De hecho, tal es la popularidad adquirida de JavaScript entre lenguajes de programación web, que aplicaciones de otros de entornos, tan conocidas como Adobe Acrobat o PhotoShop, ya recurren a su uso.

Esto demuestra que las partes buenas del lenguaje, de las que nos podemos beneficiar, compensan notablemente las partes que nos pueden conducir a cometer errores.

Se puede concluir que JavaScript es un lenguaje muy permisivo, lo que lo convierte en un lenguaje delicado, ya que al ofrecer tanta libertad al programador, éste es quien ha de asegurarse del completo funcionamiento del programa, y por lo tanto, debe conocer el lenguaje a fondo. Pero por otro lado, esta libertad es la que, conociendo las limitaciones del lenguaje y realizando un correcto uso de las características que ofrece, hace de JavaScript un lenguaje de programación enormemente expresivo, ligero y potente.

AJAX

Introducción

En el modelo clásico de aplicaciones web, cuando un usuario pulsa un botón en una página web o realiza cualquier acción que comporte la solicitud de un requerimiento HTTP al servidor, se envía la petición al servidor a la espera de obtener una respuesta

para realizar la posterior recarga de página. Esto comporta que mientras se está esperando respuesta del servidor (probablemente éste tiene que recopilar información, procesar datos…, además del tiempo que se tarda en hacer la transferencia de datos primero en un sentido y después en el otro), se detiene la aplicación web, con lo que el usuario no puede seguir interactuando con la página. Además, cuando se reciben los resultados, se procede a cargar completamente otra página HTML incluyendo los datos solicitados, lo que suele comportar un vistoso parpadeo. Todo esto ocurre prácticamente cada vez que el usuario ejecuta una acción sobre la interfaz de la aplicación, lo que supone una demora considerable y un tiempo de espera excesivo.

Este modelo fue creado como medio hipertextual. Con el tiempo y la continua evolución de tecnologías, se emplea para muchas otras cosas, como el despliegue de aplicaciones, aunque no dispone de las características idóneas para dotar a las aplicaciones web de un funcionamiento óptimo.

A menudo se dice que AJAX pretende acercar el funcionamiento y sobre todo la interactividad de las aplicaciones web al de aplicaciones de escritorio (Garret, 2005). Para ello pretende eliminar la recargas de página en cada solicitud, las esperas, los parpadeos… mediante la comunicación asíncrona con el servidor. Se trata de cargar una página, y cuando el usuario realice una acción que comporte la petición de datos al servidor, scripts y rutinas se encargarán de acceder al servidor en segundo plano, sin bloquear la aplicación, de manera que pueda seguir interactuando con el usuario sin ningún problema. Cuando se disponga de la información requerida, ésta será añadida en la porción especificada del documento sin necesidad de recargar la página, evitando incómodos parpadeos. Además, al no recargar la página completa, sino que solo se retornan los datos solicitados, éstos, al ocupar menos volumen, tardarán menos tiempo en llegar, a parte de reducir considerablemente el ancho de banda utilizado durante la conexión.

Con todo esto obtenemos una considerable mejora no sólo en la interactividad de las aplicaciones web, sino también en la velocidad de la aplicación, y todo ello influye en mejorar la usabilidad de la aplicación.

Funcionamiento

AJAX es el acrónimo de ‘Asynchronous JavaScript And XML’ (JavaScript y XML asíncrono). No se trata de un lenguaje de programación, sino de la combinación de un conjunto de tecnologías ya existentes:

- HTML o XHTML – Para la presentación del documento/página.
- CSS Cascading Style Sheets – Hojas de estilos en cascada para la definición del diseño que acompaña a la información.
- JavaScript – Es el lenguaje básico para el correcto funcionamiento de AJAX, ya que es el que se encarga de comunicar y coordinar las diferentes tecnologías, además de procesar los datos recibidos del servidor y mantener siempre la interactividad con el usuario.
- DHTML – Parser para la manipulación de objetos DOM, que nos permiten interactuar dinámicamente con la información presentada (elementos HTML).
- XML – Para el intercambio de datos con el servidor, aunque cualquier formato es válido, como por ejemplo texto plano o JSON.

- PHP – Lenguaje que se ejecuta en el servidor, aunque puede ser otro lenguaje como JSP o ASP.

En muchas páginas utilizando el modelo clásico de aplicaciones web podemos encontrar un *frame/iframe* oculto con la función de recuperar código JavaScript y/o datos del servidor sin tener que recargar la página actual. Este procedimiento cumple en toda regla con la definición de AJAX y se usa desde mucho antes que el descubrimiento de AJAX.

La mejora que desarrolla la tecnología AJAX consiste en introducir el objeto *XMLHttpRequest*, un objeto JavaScript estandarizado por el W3C (van Kesteren, 2012) que la mayoría de navegadores modernos implementan, capaz de realizar comunicaciones asíncronas con el servidor. Además, la respuesta que este objeto proporciona puede ser tratada desde JavaScript, lo que significa que se puede procesar en el cliente (navegador).

A continuación, se expone de forma resumida el funcionamiento de AJAX. Cuando ocurre un evento en el navegador que suponga la solicitud de datos al servidor, el código JavaScript incluido en el cliente debe preparar un objeto XMLHttpRequest para realizar una petición al servidor en segundo plano. Al realizar la petición por detrás de escena, el código JavaScript no tiene que detener la aplicación para esperar la vuelta de los datos solicitados, sino que espera la respuesta en segundo plano para volver a la acción cuando disponga de la totalidad de los datos requeridos (recuperación asíncrona de datos). De esta forma, el usuario no tiene que esperar durante el proceso de intercambio de información con el servidor, sino que puede seguir interactuando con la página con total normalidad. Cuando el servidor recibe la petición, procesa los datos correspondientes y envía de vuelta una respuesta con los datos solicitados a JavaScript, que procesará la respuesta obtenida para actualizar la página mostrando los cambios realizados.

Por lo tanto, la clave para el funcionamiento de AJAX es el objeto XMLHttpRequest, que es el elemento fundamental para la comunicación asíncrona con el servidor. El objeto XMLHttpRequest dispone de un conjunto de propiedades y métodos para habilitar la comunicación con el servidor a través de código JavaScript. Las propiedades más importantes son las siguientes:

- readyState: Almacena un valor que representa el estado de la solicitud enviada al servidor. Durante una conexión completa, el objeto pasa por cinco estados diferentes:
    - 0 : No inicializado, el método *open()* todavía no ha sido llamado.
    - 1 : Cargando, el método *open()* ha sido llamado.
    - 2 : Cargado, el método *send()* ha sido llamado y ya tenemos la cabecera de la petición HTTP y el *status.*
    - 3 : Interactivo, se dispone de datos parciales.
    - 4 : Completado, se dispone de la totalidad de los datos solicitados al servidor.

- onreadystatechange: En esta propiedad se debe especificar el nombre de la función *callback* que se ejecutará cuando el objeto XMLHttpRequest cambie de estado.

- responseText: Almacena la respuesta del servidor en forma de cadena.

- responseXML: Almacena la respuesta del servidor en formato XML.

- status: Almacena el código del estado de la petición HTTP. Esta propiedad sólo está disponible cuando la conexión ha superado los tres primeros estados (0,1,2). Los valores más comunes son 200, que indica que la transferencia se ha completado con éxito; 404, que indica que no hay respuesta del servidor (página no disponible); o 414, cuando los datos enviados a través del método *GET* superan los 512 bytes.

- statusText: Contiene un mensaje explicativo del estado de la petición (*status*).

Los métodos más utilizados del objeto XMLHttpRequest:

- open(): Abre un requerimiento HTTP al servidor. Se deben especificar como argumentos el método de envío de datos, la página solicitada, y si la comunicación a iniciar va a ser síncrona ('false') o asíncrona ('true') mediante un booleano. El método de envío puede ser *GET*, para peticiones pequeñas, o *POST*, cuando se ha de enviar un volumen considerable de información al servidor, ya que con el método *GET* no es posible enviar más de 512 bytes por cada solicitud.

- send(): Envía un requerimiento HTTP al servidor previamente abierto mediante el método *open()*.

- abort(): Detiene la conexión establecida con el servidor, cancelando las posibles peticiones que hubieran sido enviadas.

El objeto XMLHttpRequest presenta serias dificultades cuando tratamos de acceder a dominios externos. Los sistemas de seguridad de los navegadores no permiten el acesso de scripts a dominios distintos de donde se están ejecutando. Es un problema grave ya que si sólo se puede trabajar sobre un dominio, se reducen las posibilidades de la aplicación. No obstante, tal y como expone Holzner (2006), existen diferentes alternativas para superar esta limitación, como modificar la configuración del navegador del cliente, incluir en nuestra página espejos o *iframe's* donde mostrar el contenido de otro dominio, o se puede acceder a otro dominio utilizando código en el lado del servidor (*server-side code*) en lugar de hacerlo desde el cliente.

Conclusiones

La correcta utilización de AJAX puede enriquecer las aplicaciones web, aportando numerosos beneficios y facilidades con respecto al modelo web clásico:

- Basado en tecnologías estándares.

- Lo soportan la gran mayoría de navegadores actuales.

- Es independiente del tipo de tecnología que se utilice en el servidor.

- Es compatible con Flash.

- Portabilidad, no requiere ningún *plugin* como Flash o Applet de Java.

- Interactividad.
- Usabilidad.
- Mayor velocidad, ya que no hay que recuperar toda una página HTML en cada petición, sino simplemente los datos solicitados.
- Reduce el consumo del ancho de banda, ya que la información que se intercambia con el servidor es un conjunto de datos, y no una página completa incluyendo dichos datos.
- Mejora la estética de la página, ya que la aplicación se asemeja a una de escritorio.

No obstante, hay algunos aspectos a tener en cuenta:

- Aunque el objeto XMLHttpRequest ya es un estándar, en cada navegador funciona de forma distinta, así que hay que asegurarse de escribir código que sea válido para, como mínimo, los navegadores más importantes (Zaera, 2006).
- Pueden surgir dificultades con navegadores antiguos que no implementen esta tecnología.
- Puede haber clientes con el JavaScript deshabilitado, lo que impediría la ejecución de los scripts de la aplicación.
- Se pierde el concepto de volver a la página anterior, ya que esta acción nos lleva a la última página cargada, y utilizando AJAX, realizar una acción, la mayoría de veces no implica una recarga de página, sino la actualización de alguna porción de ésta, de manera que al volver a página anterior no iríamos a la última actualización de página, sino a la última recarga de página, y en AJAX, generalmente no es lo mismo.
- Las páginas que utilizan AJAX no se comportan de la misma forma que las páginas que no lo emplean. En la red podemos encontrar todo tipo de páginas, y esto puede ser un inconveniente ya que la existencia de páginas con y sin AJAX puede provocar confusión en los visitantes.
- Requiere programadores que conozcan todas las tecnologías que intervienen en AJAX.
- No es posible conectarse con un objeto XMLHttpRequest a un dominio distinto de donde se ha creado el objeto, por lo que nos obliga a tener toda la aplicación bajo el mismo dominio.

Aunque todavía presenta demasiados inconvenientes, éstos son en cierto modo subsanables, y las ventajas que ofrece son considerables con respecto a otras tecnologías. Una prueba es la variedad de aplicaciones interactivas que se pueden programar con AJAX de forma práctica y eficaz, como afirma Holzner (2006), como búsquedas en tiempo real (tal y como escribe el usuario se genera la búsqueda y una lista de resultados sin recarga de página), obtener posibles respuestas a la entrada de un usuario mediante un autocompletado, salas de chat, juegos, *dragging and dropping*

(añadir o eliminar elementos de una lista mediante su desplazamiento con el ratón - arrastrándolos), configuración de menús *pop-up* (al pasar por encima de un vínculo con el ratón se extiende una lista con varias opciones), paginación (cuando el contenido no cabe en una página y se tiene que dividir en varias accesibles mediante hiperenlaces, al cambiar de página, ésta no se recarga completamente, simplemente actualiza su contenido), obtener retroalimentación instantánea en intentos fallidos de registro, actualización inmediata de páginas web (sin recargas), o aplicaciones tan conocidas como *Google Maps*.

Yahoo! User Interface Library

Yahoo (Yahoo! Inc., 2012), proporciona una colección de librerías y utilidades implementadas en JavaScript para añadir nuevas funcionalidades y widgets a la Web a través de código JavaScript, o bien para simplificar el uso y la manipulación de objetos y funcionalidades ya existentes.

Las distintas funcionalidades están organizadas en módulos, de manera que cada módulo representa una utilidad o un widget instanciable dentro de código JavaScript mediante un objeto YUI. Cada módulo contiene una serie de clases, donde cada clase posee sus atributos, métodos y eventos, que permiten la manipulación de los objetos YUI.

Durante el desarrollo de este proyecto, se han empleado diversos módulos de YUI Library para simplificar algunos aspectos como la manipulación del DOM, o para agregar algunas funcionalidades, como la función de autocompletado en algunas entradas de los formularios.

En un principio, se utilizaron varios módulos de la segunda versión de YUI. Sin embargo, durante el desarrollo del proyecto, Yahoo lanzó una tercera versión de YUI, reparando los posibles errores en las utilidades existentes y añadiendo algunas nuevas; de manera que se modificó todo el código del proyecto relativo a YUI Library para actualizarlo a la nueva versión, y, obviamente, todo el código de YUI Library añadido a posteriori pertenece a la última versión.

Aunque se han utilizado varios módulos a lo largo del proyecto, se expone una explicación del funcionamiento de los principales módulos empleados, es decir, los más utilizados y más relevantes para el funcionamiento de la aplicación. Hay una serie de módulos que se han usado durante el desarrollo del proyecto pero no se detallan en este apartado, ya que se han incluido para acciones simples y puntuales, y no requieren una extensa explicación, y además, se verán más adelante durante la sección de desarrollo. Un ejemplo es el módulo *json*, que proporciona métodos para tratar cadenas JSON como objetos JavaScript, o distintos *Listener's*, que permiten realizar escuchas sobre entradas o elementos HTML para un conjunto variado de eventos.

YUI 2: AutoComplete

El primer paso para incluir cualquier módulo de YUI en nuestro código es añadir las dependencias, es decir, incluir el archivo fuente que contiene la implementación de las utilidades. Cada archivo fuente corresponde a un módulo, de forma que se tienen que añadir tantos archivos fuente como módulos se pretendan utilizar.

El módulo *autocomplete* proporciona propiedades y métodos para facilitar la implementación de la función autocompletar sobre una entrada de texto HTML. Es decir, cuando el usuario teclee sobre una entrada con AutoComplete, se expandirá un contenedor con una lista de resultados, obtenidos de un conjunto de datos previamente asignado, y filtrados en función de la cadena introducida.

Para instanciar un objeto AutoComplete es necesario asignarle la entrada donde trabajará y el conjunto de datos con el que realizará las comparaciones. Para el correcto funcionamiento del widget, el elemento HTML asignado como entrada tiene que disponer de una estructura específica: un contenedor ( ) que contenga dos elementos, una entrada de texto, que es donde escribirá el usuario, y otro contenedor, que es donde se depositará la lista de resultados filtrados.

El conjunto de datos asignado al objeto AutoComplete tiene que ser una instancia de DataSource, que es un objeto YUI para representar fuentes de datos. Aunque se pueden definir varios tipos de fuentes de datos, durante el desarrollo del proyecto se ha utilizado una fuente de datos local (de YUI 2), generada a partir de un vector con los resultados.

Generando los tres elementos mencionados ya se dispone de una entrada con auto-completado que funcione correctamente. Cuando el usuario escriba sobre la entrada de texto indicada, AutoComplete buscará en el conjunto de datos asignado los datos que comiencen por la cadena introducida y mostrará el subconjunto obtenido en el contenedor en forma de lista. Además, si el usuario selecciona unos de los resultados mostrados, éste será depositado en la entrada de texto asignada y el contenedor volverá a ocultarse de nuevo.

Cada objeto AutoComplete dispone de numerosos atributos modificables para personalizar la configuración del widget. Se pueden destacar los más utilizados:

- minQueryLength: Número mínimo de caracteres a introducir para iniciar la consulta. Su valor por defecto es 1.

- queryDelay: Tiempo de espera después de haber introducido un carácter hasta lanzar la consulta. Su valor por defecto es 0.1 (segundos).

- maxResultsDisplayed: Número máximo de resultados que puede contener la lista del contenedor. Su valor por defecto es 10.

- autoHighlight: Permite remarcar el primer elemento de las lista de resultados. Su valor por defecto es ‘true’.

- forceSelection: Permite exigir al usuario escoger un resultado de la lista del contenedor, de lo contrario el valor de la entrada es eliminado. Su valor por defecto es ‘false’.

- responseSchema: Permite definir varios campos para cada resultado.

En el instante en que se introduce un carácter en la entrada de texto asignada, AutoComplete envía la consulta correspondiente a la fuente de datos indicada en busca de resultados. No obstante, se puede modificar el contenido de las peticiones redefiniendo el método *generateRequest()*.

Cada instancia de AutoComplete ofrece un conjunto de eventos a los que subscribirse en diversos momentos o situaciones. Algunos de los más destacados son:

- dataReturnEvent: Se ha recibido un conjunto de resultados de la fuente de datos especificada.
- itemSelectEvent: Un elemento de la lista de resultados del contenedor ha sido seleccionado.
- unmatchedItemSelectEvent: Se ha seleccionado una entrada no coincidente con ningún elemento de la lista de resultados del contenedor.
- textBoxChangeEvent: La entrada de texto ha cambiado su valor y ha perdido el foco.
- textBoxKeyEvent: La entrada de texto ha recibido algún carácter o símbolo mediante el teclado.

YUI 3: Global Object

El primer paso para incluir utilidades de YUI Library, tal y como se ha explicado anteriormente, es cargar todas las dependencias. Este es uno de los aspectos que pretende mejorar YUI 3, que aunque también permite realizar un cargado estático, proporciona herramientas para un cargado dinámico mediante un único archivo: el archivo raíz de YUI. Este archivo contiene el código necesario para cargar dinámicamente las dependencias requeridas sobre la demanda, además de los módulos básicos para el funcionamiento de cualquier objeto YUI (YUI Global Object, Node, Event).

El archivo raíz añade una única variable global a la página: el objeto YUI. Para utilizar cualquier utilidad de YUI, se debe crear una instancia del objeto global YUI donde se especifican los módulos a utilizar. Para especificar los módulos pertinentes, se utiliza el método *use()*, donde se pasan como argumentos los módulos requeridos en forma de cadena, tantos como sean necesarios; y una función *callback*, que se ejecutará una vez hayan sido cargados todos los módulos indicados y podrá usar las utilidades incluidas en los módulos cargados.

Este patrón recibe el nombre de *sandbox*, y es la base para poder usar las utilidades de YUI en su última versión. No sólo facilita el cargado de dependencias, sino que asegura la correcta estructuración y organización del código, de manera que nunca contaminará el alcance global de la página, ni interferirá con otros objetos globales de JavaScript que puedan estar en uso. De la misma forma, se pueden tener múltiples *sandboxes* en la misma página sin generar ningún tipo de conflicto. La estructura básica de un *sandbox* de YUI 3 es la siguiente:

```
YUI().use('module1', 'module2', 'module3', function (Y) {

        /* Código JavaScript */
        /* Métodos, atributos y eventos de 'module1',
        'module2' y 'module3' habilitados */
});
```

Además, el objeto global YUI proporciona un conjunto de clases donde se definen distintas utilidades, de las cuales se pueden destacar:

- YUI: Constructor para instancias del objeto global YUI.
- rls (Remote Loader Service): Soporte para el cargado dinámico de módulos.
- config: Proporciona un conjunto de propiedades modificables para la configuración de la instancia YUI.
- Get: Permite obtener y/o cargar dinámicamente scripts y elementos DOM.
- Array: Proporciona un conjunto de operaciones de array.
- Lang: Proporciona un conjunto de métodos JavaScript.
- Object: Proporciona un conjunto de utilidades para la manipulación de objetos JavaScript.

YUI 3: Node

Se trata de una interfaz para la manipulación de elementos DOM. De hecho, la API de Node está basada en la API estándar del DOM, aportando algunas propiedades y métodos adicionales para facilitar las operaciones más comunes.

El módulo *node* contiene dos clases: *Node* para tratar con nodos independientes; y *NodeList*, para tratar con conjuntos de nodos.

El método básico para la obtención de nodos es *one()*, y acepta como parámetro tanto elementos DOM como selectores CSS. Se puede aplicar tanto sobre el objeto global de YUI, como sobre una instancia de Node, y devolverá siempre el primer elemento coincidente que encuentre. Aplicado sobre el objeto global (Y) buscará elementos por todo el documento. En cambio, aplicado sobre una instancia Node, buscará elementos entre los descendientes del nodo referido. Para obtener un conjunto de nodos, Node proporciona el método *all()*, que devuelve un objeto NodeList.

Para crear nuevos nodos se dispone del método *create()*. Una vez se ha creado una instancia de Node, existen varios métodos para la manipulación básica del nodo:

- append: Permite agregar contenido, que sitúa como último hijo del nodo.
- prepend: Permite agregar contenido, esta vez como primer hijo del nodo.
- setContent: Permite reemplazar el contenido.
- insert: Permite insertar contenido.

Es posible acceder a las propiedades o atributos de los nodos mediante los métodos *get()* y *set()*, para obtener o para especificar sus valores, respectivamente.

El método *on()* permite añadir escuchas a cualquier instancia de Node. Se aplica sobre la instancia Node y se le asignan como parámetros una cadena indicando el evento que se pretende escuchar, y una función que se ejecutará una vez se cumpla la acción

indicada sobre el nodo asociado. Este método dispone de una larga lista de eventos DOM a los que suscribirse.

YUI 3: Event

El módulo *event* proporciona las herramientas necesarias para realizar escuchas sobre un elemento HTML a la espera de una acción previamente especificada. Se realiza mediante el método *on(type, callback)*, donde 'type' es el tipo de evento y 'callback' la función a ejecutar en el momento en que se produzca el evento. Este método se puede aplicar sobre instancias de Node y NodeList y permite detectar una amplia lista de eventos DOM.

El método *on()*, igual que cualquier otro método de suscripción, devuelve un objeto de suscripción que se puede usar para deshacer dicha suscripción aplicando el método *detach()* sobre dicho objeto.

Hay varias formas de suscribir un elemento a un evento:

- Suscribirse desde una instancia de Node.
- Suscribirse desde una instancia del objeto global de YUI (Y). En este caso, el método *on()* requiere un argumento extra, el selector CSS del nodo asociado.
- Suscribirse solamente la primera vez que tenga lugar el evento: método *once()*.
- Suscribirse a ser notificado después de que se produzca el evento: método *after()*. Con el método *on()*, la notificación se realiza en el mismo momento en que se produce el evento, antes de que éste tenga consecuencias. En cambio, con el método *after()*, la notificación se envía después de que haya tenido lugar el evento y éste haya desencadenado las acciones correspondientes.
- Suscribir un elemento a diversos eventos pasándole al método *on()* un array o un objeto como argumento. En el caso de pasarle un array, se pueden especificar diversos tipos de evento para el mismo nodo. En el caso de pasarle un objeto, además, para cada tipo de evento se puede especificar una función *callback* distinta. Esta configuración permite deshacer todas las suscripciones vinculadas a través del vector o el objeto al nodo indicado en una sola llamada.

Algunos eventos tienen un comportamiento habitual, como el evento *submit*, que envía los datos contenidos en un formulario y realiza una petición de página al servidor. Para cancelar este tipo de comportamiento, Event proporciona el método *preventDefault()*.

La mayoría de eventos se pueden escuchar desde el elemento concreto que lo origina, o bien desde cualquier elemento padre. Es decir, cuando se suscribe un elemento a un evento, se ejecutará la función *callback* siempre que se produzca el evento en el elemento asociado o en cualquiera de sus hijos. Para evitar este comportamiento, Event dispone de un método *stopPropagation()*, que evita que un elemento escuche eventos asociados a otros elementos superiores en la estructura/jerarquía del DOM.

Para suspender la ejecución de todos los eventos suscritos a un elemento, se dispone de los métodos *stopImmediatePropagation()* y *halt(true)*.

El módulo *event-synthetic* proporciona la infraestructura necesaria para la creación de nuevos eventos a partir de eventos DOM ya existentes. YUI 3 proporciona un conjunto

de módulos con eventos sintéticos: *event-flick*, *event-focus*, *event-gestures*, *event-hover*, *event-key*, *event-mouseenter*, *event-mousewheel*, *event-move*, *event-outside*, *event-resize*, *event-touch* y *event-valuechange*. Se suscriben y se comportan de la misma forma que cualquier otro evento DOM.

YUI 3: AutoComplete

Se trata de la misma utilidad expuesta anteriormente, aunque de una versión posterior. El fin es el mismo, no obstante se han mejorado algunos aspectos y para ello se ha modificado su funcionamiento y metodología.

Existen dos formas de instanciar un objeto AutoComplete: como *plugin* y como clase. Para instanciar un AutoComplete como *plugin*, se usa el método *plug()* para adjuntar la nueva instancia (*Y.Plugin.AutoComplete*) a una instancia ya existente de Node. Una vez instanciado el AutoComplete, se puede acceder a él a través de la propiedad *ac* del nodo asignado.

Para instanciar AutoComplete como una clase, se crea una instancia a través del objeto global (*Y.AutoComplete)* donde es necesario especificar, a través de un objeto de configuración, una entrada para el AutoComplete, que puede ser un selector CSS, una instancia de Node, o un elemento DOM en referencia a un elemento HTML tipo <input> o <textarea>. Esta vez, a diferencia que en la versión anterior, no es necesario definir una estructura con la entrada de texto y el contenedor, sino que simplemente se debe asignar una entrada o nodo válido, y el propio AutoComplete se encarga de generar los elementos necesarios.

La fuente de datos que se le asigna a la instancia AutoComplete como conjunto de resultados para que compare con el valor de la entrada puede ser de diversos formatos: array/objeto, instancia de DataSource, función JavaScript, JSONP/XHR URL, instancia de Node incluyendo una entrada tipo <select>, o una consulta YQL.

Aunque AutoComplete dispone de una extensa lista de atributos de configuración, se comentan los más destacados:

- inputNode: Es el único atributo requerido para crear una instancia. Elemento tipo <input> o <textarea> para monitorizar los cambios.

- source: Fuente de datos que usará el AutoComplete como conjunto de datos.

- maxResults: Número máximo de resultados que puede contener la lista del contenedor. Un valor igual o inferior a 0 implica un máximo ilimitado de resultados. Su valor por defecto es 0.

- minQueryLength: Número de caracteres que deben ser introducidos antes de lanzar la consulta. Un valor de 0 habilita las consultas vacías, en cambio, un valor negativo deshabilita cualquier evento de consulta, comportando la conclusión del funcionamiento del AutoComplete. Su valor por defecto es 1.

- queryDelay: Tiempo de espera después de introducir un carácter hasta lanzar la consulta. Su valor por defecto es 100 (ms).

- requestTemplate: Es el patrón de consulta. Es modificable y puede ser una función que acepta una consulta (el valor de la entrada, lo que haya escrito el usuario) como parámetro y devuelve una cadena, o una cadena que contenga

el parámetro '{query}', el cual será sustituido por la consulta codificada. La cadena resultante se adjuntará a la petición URL cuando el atributo *source* sea una fuente de datos remota, JSONP URL o XHR URL.

- resultFormatter: Es una función modificable que permite definir un formato personalizado para los elementos de la lista del contenedor.

- resultFilters: Permite escoger el tipo de filtrado.

- resultHighLighter: Permite escoger el tipo de remarcado.

- resultListLocator: Cadena localizadora o función que sirve para extraer el vector de resultados en respuestas que no tengan formato de array.

- resultTextLocator: Cadena localizadora o función que sirve para extraer texto plano en resultados que no tienen formato de cadena.

- activateFirstItem: Permite activar automáticamente el primer elemento de la lista cuando se expande el contenedor o bien cuando cambian los resultados. Su valor por defecto es 'false'.

- circular: Permite activar/desactivar la navegación circular a través de la lista de resultados. Es decir, el primer elemento se toma como el siguiente al último, y de la misma forma, el último elemento se considera el anterior al primero. Su valor por defecto es 'true'.

- scrollIntoView: Permite añadir una barra de desplazamiento cuando el tamaño del contenedor no alcanza para mostrar todos los elementos de la lista de resultados. Su valor por defecto es 'false'.

A veces, la respuesta que se recibe de una consulta es una estructura desconocida para AutoComplete, posiblemente un objeto con más metadatos a parte del resultado, de manera que no sabrá donde localizar los resultados dentro de dicha estructura. Para especificarle la ruta de acceso o las propiedades a las que tiene que acceder para hallar el vector de resultados dispone del atributo *resultListLocator*. De la misma forma que el conjunto de resultados puede llegar en forma de estructura de datos compleja, ocurre algo similar con los resultados. Cada elemento del vector de resultados puede ser un objeto y no una simple cadena. Para definir la ruta de acceso para encontrar la solución dentro de un objeto resultado se dispone del atributo *resulTextLocator*.

Añadiendo el módulo *autocomplete-filters* se habilita la propiedad *resultFilters* para el filtrado de soluciones. Funciona especificándole un método de filtrado de entre un conjunto de métodos disponibles:

- charMatch: Devuelve resultados que contengan todos los caracteres introducidos, en cualquier orden.

- phraseMatch: Devuelve resultados que contengan la cadena introducida.

- startsWith: Devuelve resultados que comiencen por la cadena introducida.

- subWordMatch: Devuelve resultados en los que todas la palabras completas coincidan con palabras completas o bien con una parte de las palabras en el resultado.

- wordMatch: Devuelve resultados que contengan todas las palabras completas de la consulta, en cualquier orden.

- _Case: Es un sufijo que seguido de cualquiera de los cinco anteriores métodos proporciona un nuevo método. Por defecto, las consultas no distinguen entre mayúsculas y minúsculas. Añadiendo 'Case' a cualquier filtro se obtiene un nuevo filtro que funcionará de la misma forma pero incluyendo sensibilidad a las mayúsculas.

- _Fold: Es un sufijo que seguido de cualquiera de los cinco anteriores métodos proporciona un nuevo método. Por defecto, las consultas distinguen caracteres acentuados. Añadiendo 'Fold' a cualquier filtro se obtiene un nuevo filtro que funcionará de la misma forma pero ignorando cualquier tipo de acento.

Para resaltar los resultados o parte de ellos, normalmente la parte coincidente con la consulta, utilizaremos el atributo *resultHighLighter* del módulo *autocomplete-highlighters*. Dispone de los mismos filtros que *resultFilters* y funciona completamente igual, excepto que en lugar de utilizar los filtros para realizar la consulta, los utiliza para remarcar los resultados, es decir, la parte de los resultados que coincide con la consulta (la forma en que coincide depende del filtro) la muestra en negrita.

También se pueden crear filtros personalizados. De hecho, *resultFilters* no es más que una función que recibe la consulta actual y un vector de resultados y devuelve un vector de resultados filtrados. Para crear un filtro propio, se debe declarar una función '*customFilter*' y asignársela al atributo *resultFilters*.

AutoComplete proporciona una interesante lista de eventos con los que manipular la información en diferentes fases del procedimiento. Los más frecuentes son:

- query: El valor de la entrada de texto ha cambiado y reúne las condiciones necesarias para lanzar una consulta.

- results: Se ha recibido un conjunto de resultados de la fuente de datos especificada.

- select: Un elemento de la lista de resultados del contenedor ha sido seleccionado.

YUI 3: DataSource

El objeto DataSource es de utilidad para la representación de fuentes de datos y para su posterior manipulación.

Se pueden crear distintos tipos de fuentes de datos, pero podemos distinguir básicamente entre fuentes locales y fuente remotas. En las fuentes de datos locales (*Y.DataSource.Local*) los datos se encuentran almacenados en la memoria local, normalmente se trata de un array o un objeto JavaScript. En cambio, las fuentes de datos remotas acceden a datos almacenados en un servidor web. Hay tres tipos de instancias de DataSource remotas:

- Y.DataSource.Get: Accede al servidor mediante la utilidad *Get*. Soporta la recuperación de datos para recursos almacenados en distintos dominios sin necesidad de un *proxy*, pero el servidor debe retornar los datos en JSON y soportar un parámetro 'script callback' para devolver adecuadamente la respuesta. Este parámetro contiene el nombre de la función a la que serán enviados los datos resultantes cuando sean devueltos.

  Adjuntar el *plugin DataSourceJSONSchema* a la instancia DataSource permite normalizar los datos que se enviarán a la función *callback* a través del objeto de configuración *schema*. El atributo *resultListLocator* permite indicar donde se encuentra el vector de datos dentro del objeto JSON, y el atributo *fields* permite especificar los campos que contiene cada objeto/dato.

- Y.DataSource.IO: Accede al servidor mediante la utilidad *IO,* que permite establecer comunicación asíncrona con el servidor. Para acceder a un servidor entre dominios se requiere un agente *proxy* del mismo dominio o bien habilitar la característica XDR de *IO* para superar las restricciones de seguridad del navegador. La utilidad *IO* soporta la recuperación de datos de diversos formatos, entre ellos JSON, XML y texto plano.

  En caso necesario, se debe adjuntar el *plugin* de *DataSchema* adecuado para normalizar los datos. Para ello, DataSource dispone de las siguientes clases, que deben usarse en función del formato de respuesta del servidor: *DataSourceArraySchema*, *DataSourceJSONSchema*, *DataSourceXMLSchema* y *DataSourceTextSchema*.

- Y.DataSource.Function: Definir una función propia JavaScript que devuelva datos dada una petición permite una personalización completa del mecanismo de recuperación de datos.

DataSource dispone de un conjunto de eventos interesantes a los que suscribirse. Los más comunes:

- request: Se ha enviado una petición.

- data: Se han recibido los datos, antes de ser tratados.

- response: Se ha devuelto la respuesta a la función callback correspondiente.

Estudios realizados para hacer el trabajo

Esta sección incluye el conjunto de tecnologías relacionadas con la aplicación Rhizomer que ha sido necesario estudiar antes de empezar a desarrollar el trabajo, ya sea porque la elaboración del proyecto requiere su uso, o bien porque la aplicación sobre la que se trabaja está basada en algunas tecnologías características, lo que hace imprescindible asimilar sus nociones básicas para entender el funcionamiento y el objetivo de Rhizomer.

La mayoría de las tecnologías incluidas en esta sección coinciden con las tecnologías explicadas en la sección anterior, por lo que se trata de una simple enumeración.

Aunque se ha buscado información en varios libros, artículos y sitios web, se expone junto a cada tecnología estudiada la fuente de información que sirvió como base para entender cada lenguaje.

- JavaScript (Crockford, 2008).
- HTML (W3C, 1999).
- AJAX (Holzner, 2006).
- Web Semántica (Pollock, 2009).
- RDF (Pollock, 2009).
- Ontologías y OWL (Pollock, 2009).
- SPARQL (W3C, 2008a).

Otros trabajos similares

- http://www.freebase.com/docs/suggest

Freebase contiene más de diez millones de recursos semánticos, distinguiendo entre más de 3.000 tipos con más de 30.000 propiedades. Se trata de una base de datos considerable. Esta herramienta tiene varios aspectos similares al proyecto desarrollado.

Ofrece una interfaz donde lanzar consultas sobre el repositorio local de datos para la localización de recursos a través de una entrada con auto-completado. También permite la edición de recursos, así como asiste al usuario durante el proceso de edición mostrando valores posibles mediante un mecanismo de auto-completado en función del valor introducido y de los recursos disponobles en el repositorio local de datos. Aquí encontramos un aspecto mejorado durante el proyecto, y es que para obtener los resultados mostrados durante la modificación del valor de una propiedad del recurso en edición, la restricción se hace en función de la cadena escrita por el usuario únicamente, indistintamente del tipo de cada dato. En cambio, el mecanismo de autocompletar implementado durante el proyecto comprueba, en acciones similares a la comentada, que el tipo del recurso coincida con el rango de la propiedad a la que pertenece para mantener la consistencia del conjunto de datos, además de realizar las comparaciones correspondientes entre la cadena introducida y los nombres/etiquetas de los recursos.

Además, esta aplicación también permite al usuario definir una nueva descripción semántica en caso de que los resultados mostrados por el mecanismo de auto-completado durante el proceso de edición de un recurso no satisfagan sus requerimientos y quiera definir un valor nuevo.

Otro aspecto suplementario respecto a Freebase que se ha aportado durante el proyecto es la posibilidad de realizar las consultas del mecanismo de autocompletar en un servidor externo (en nuestro caso, DBpedia, aunque es fácilmente configurable para asignar otro servidor), de tal forma que cuando un usuario está editando la descripción de un recurso, si no obtiene el valor deseado de entre los recursos contenidos en el repositorio local de datos, tiene la opción de buscar en otro conjunto de datos.

- http://www.mediawiki.org/wiki/Extension:Semantic_Forms

MediaWiki es un software libre que utiliza PHP para procesar y mostrar datos almacenados en la base de datos. Una Wiki es una página web cuyo contenido puede ser editado por múltiples usuarios a través de cualquier navegador de forma sencilla. Dichas páginas se desarrollan a partir de la colaboración de los internautas, quienes pueden agregar, modificar o eliminar información.

Existe una extensión para MediaWiki llamada ‘Semantic Forms’ que facilita la manipulación de datos semánticos. Permite consultar, editar o añadir contenido mediante formularios. Los usuarios pueden crear y editar los formularios sin necesidad de escribir código. Esta extensión también soporta el autocompletado en los campos de los formularios, aunque funciona con un repositorio de datos locales, es decir, los datos están almacenados directamente en la página. No obstante, se puede configurar

para que se envíen las consultas al servidor local, de forma que se realice cada petición de datos en tiempo real.

En nuestro caso, las consultas para obtener los elementos resultantes para el mecanismo de auto-completado son siempre en tiempo real, ahorrándonos la necesidad de almacenar grandes conjuntos de datos en nuestra página.

Otro aspecto a destacar es que nuestra aplicación, a diferencia de MediaWiki, durante la edición de un recurso, permite realizar las consultas de 'autocomplete' en otro dominio al que pertenece la aplicación, dándole al usuario la posibilidad de buscar en otras fuentes si en los resultados obtenidos de nuestro repositorio no se encuentra lo que busca.

- http://ontowiki.net/Projects/OntoWiki

Esta aplicación es muy similar, especialmente la parte que se trata durante el proyecto. Empezando por la base de datos, está representada en RDF para una óptima definición y representación de ontologías y metadatos. Consecuentemente, el motor de consultas y recuperación de datos también está basado en SPARQL. Además, la forma de organizar las diferentes clases de datos y ontologías en la interfaz también es semejante, estructurados en facetas que permiten el filtrado de los datos.

Para la edición de datos también posee formularios fácilmente editables y con asistencia al usuario mediante entradas con autocompletar para informar de los recursos factibles en función del dominio del recurso a especificar, de manera que no afecte a la consistencia del sistema.

Las dos aplicaciones muestran los datos de manera enlazada. Es decir, se puede acceder a la descripción semántica de las propiedades o de sus valores con un simple click (siempre y cuando representen un recurso y no un simple valor literal), facilitando de esta forma la navegación por todos los elementos del conjunto de datos.

Otra característica en común entre las dos herramientas es que permiten la creación de nuevas descripciones semánticas para el caso en que se quiere asignar a alguna propiedad un valor no existente en las ontologías definidas.

El último factor a comparar es que ambas aplicaciones pueden acceder a repositorios externos en busca de datos que no se encuentren en el repositorio local.

En las siguientes capturas de pantalla se muestran algunas de las funcionalidades de las que dispone OntoWiki similares al trabajo realizado durante el desarrollo del proyecto: se muestra cómo editar la descripción semántica de un recurso mediante la adición de una propiedad y la asignación de su respectivo valor, todo el proceso asistido mediante un mecanismo de auto-completado que ofrece valores existentes o de conjuntos de datos referencia.

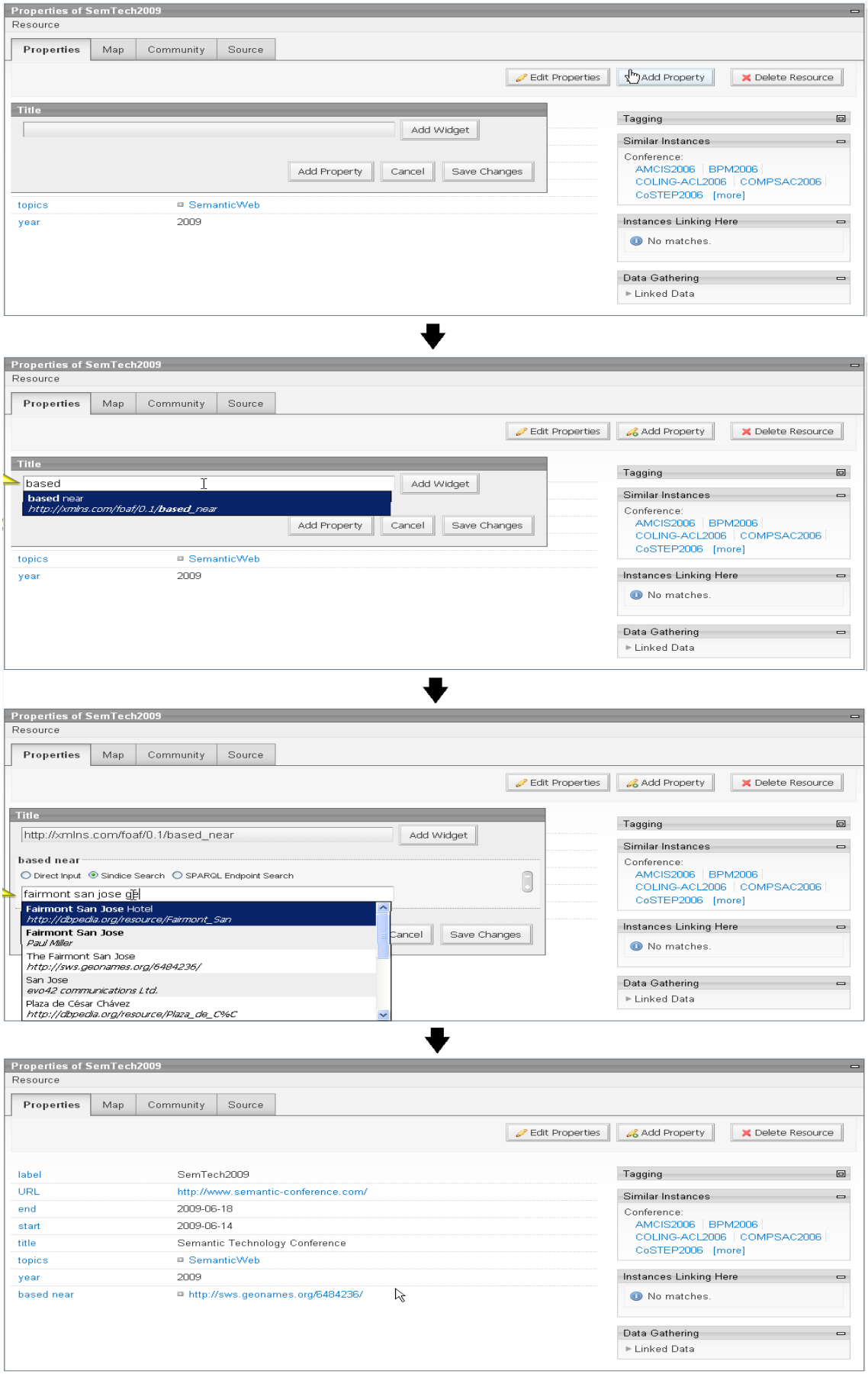

Figura 1: OntoWiki

Bloque de desarrollo

Gestión del proyecto

Por las características del proyecto realizado, ya que se trata de mejorar algunas funcionalidades de una aplicación existente y añadir nuevas, seguiremos un modelo de desarrollo iterativo, donde podemos desglosar en iteraciones el trabajo hecho a lo largo del proyecto.

De esta forma, cada iteración expuesta corresponde a una funcionalidad añadida o a una modificación realizada sobre un aspecto concreto de la aplicación, de modo que queda separado y organizado el trabajo realizado en distintas secciones o apartados de la aplicación.

Además, las iteraciones están ordenadas cronológicamente, es decir, en el mismo orden en que se han ido implementando.

Podemos distinguir entre dos etapas claramente diferenciadas durante el desarrollo del proyecto:

Etapa 1

- Trabajo previo.

Estudio de las tecnologías relacionadas con la aplicación Rhizomer y de los lenguajes a utilizar durante el desarrollo del proyecto - JavaScript, AJAX, Web Semántica, HTML.

- Iteración 1: Formulario genérico de consulta.

Se trata de implementar un formulario HTML para localizar recursos contenidos en el repositorio local de datos, almacenados en formato RDF, mediante la especificación de algunas de las propiedades genéricas de los recursos.

- Iteración 2: Intérprete SPARQL para el formulario Web.

Se trata de implementar una función que transforme un formulario total o parcialmente rellenado en una consulta SPARQL, restringiendo los valores indicados en el formulario para cada propiedad.

- Iteración 3: Formulario específico de consulta.

Dado un tipo concreto de datos, se trata de generar automáticamente un formulario en función de las propiedades específicas para el tipo indicado. El formulario está destinado a la localización de recursos dentro del subconjunto de datos seleccionado.

- Iteración 4: Formulario dinámico de consulta.

Se trata de ofrecer la posibilidad al usuario de añadir propiedades genéricas al formulario específico. Es decir, se permite añadir nuevas entradas al formulario donde especificar el valor de las propiedades genéricas establecidas.

- Iteración 5: AutoComplete.

El método para añadir propiedades consistirá en una caja de texto con un sistema de

auto-completado, de forma que al escribir sobre la entrada, se habilitará una lista de las propiedades genéricas contenidas en el conjunto de datos local cuyo nombre/etiqueta coincida en su inicio con la cadena introducida. De esta forma, el usuario puede seleccionar una de las propiedades existentes antes de añadir la entrada respectiva.

- Iteración 6: Diálogo de confirmación.

Se trata de añadir un cuadro de diálogo que se muestre cuando el usuario decida añadir una nueva entrada con la finalidad de confirmar la acción seleccionada.

Intervalo entre etapas 1 y 2

Debido a motivos personales, decidí aparcar la elaboración del proyecto durante dos o tres meses (durante un verano). Por razones que escapan de mi poder, y muy a mi pesar, los dos meses se convirtieron en dos años. Lógicamente, durante este largo período de tiempo, la aplicación sobre la que trabajo, Rhizomer, ha ido evolucionando, de forma que ha sufrido notables cambios. No obstante, de las seis primeras iteraciones, todavía se conservan algunas características.

El formulario de búsqueda ha desaparecido y en su lugar se ha implementado un sistema de navegación y localización de recursos basado en menús de navegación y facetas. Una vez se selecciona un recurso a través de las facetas, obtenemos una vista específica del recurso donde se permite, entre otras cosas, editar el recurso seleccionado. Para la edición del recurso se proporciona un formulario dinámico basado, en algunos aspectos, en el antiguo formulario de búsqueda. Por ejemplo, el formulario de edición, igual que el de consulta, se genera automáticamente. La diferencia es que el primero lo hacía en función de las propiedades específicas de un tipo concreto de datos, en cambio, el de edición se genera en función de las propiedades definidas para el recurso indicado. El formulario de edición también ofrece la posibilidad de añadir entradas adicionales para definir nuevas propiedades, y también funciona con un sistema de auto-completado. La funcionalidad de autocompletar se ha ampliado para que también ofrezca soporte a las entradas para los recursos, es decir, a las cajas de texto que corresponden a los valores de las propiedades pero que no contienen valores literales.

Hay que dejar claro que el proyecto se basa principalmente en la segunda etapa del desarrollo, ya que las funcionalidades y servicios introducidos durante la primera etapa han quedado obsoletos. De todas formas, creo que es necesario especificar en la sección de desarrollo el trabajo hecho durante la primera etapa para entender los sucesivos pasos tomados durante el proyecto, a parte de aclarar que las funcionalidades principales de los formularios de edición implementados durante mi ausencia están basadas en parte del trabajo hecho durante la primera etapa.

Durante la segunda etapa, el trabajo se centra en añadir algunas características y funcionalidades a los formularios de edición recientemente comentados, de manera que para situarnos en el contexto de trabajo, entendemos que partimos de un formulario de edición de un recurso específico, que contiene como campos todas las propiedades definidas con sus respectivos valores editables.

Etapa 2

• Iteración 7: Formulario para la definición de descripciones semánticas.

Se trata de generar de forma automática un formulario cuando el usuario confirme el valor editado de una propiedad de tal forma que éste no corresponda a ningún recurso existente en el repositorio local de datos. El formulario está destinado a definir algunas propiedades para el recurso recién especificado.

• Iteración 8: Confirmar valor mediante 'enter'.

Debido a un error en el funcionamiento de las librerías utilizadas para la implementación del AutoComplete, sólo se puede acceder al nuevo formulario cuando la actual entrada pierde el foco, ya sea mediante el tabulador o bien a través del ratón. En cambio, al pulsar la tecla ‘intro’ no considera que se esté confirmando el valor introducido. Esta parte consiste en habilitar la tecla ‘intro’ del teclado para que acepte la cadena introducida en la entrada de texto durante el proceso de edición del valor de una propiedad y, en caso necesario, se genere el formulario para la descripción semántica.

• Iteración 9 : De YUI 2 a YUI 3.

Se trata de modificar el código respectivo a ‘Yahoo! User Interface Library’ incluido en el proyecto para actualizarlo a la nueva versión: YUI 3 Library.

• Iteración 10: Trabajar con datos en JSON.

Se trata modificar el mecanismo de consulta para cambiar el formato de respuesta de RDF a JSON, y como consecuencia, modificar los métodos que traten con datos recibidos del servidor.

• Iteración 11: AutoComplete en tiempo real.

Hasta ahora, el AutoComplete implementado para la entrada de las propiedades funcionaba con una fuente de datos local. Modificar los aspectos necesarios para conseguir un AutoComplete que funcione con una fuente de datos remota.

• Iteración12: Nueva descripción semántica como primer resultado de AutoComplete.

Hasta ahora, se generaba un formulario para definir una descripción semántica sólo en el caso que se aceptara una cadena introducida no coincidente con ningún recurso existente en el repositorio local de datos como valor de una propiedad editable. Se pretende mostrar siempre como primer elemento de la lista de resultados del AutoComplete la posibilidad de definir una nueva descripción.

• Iteración 13: Consultas de AutoComplete en fuentes de datos externas.

Se trata de ofrecer la posibilidad de modificar la base de datos asignada a AutoComplete para que éste realice las consultas pertinentes sobre un servidor externo, en lugar de continuar buscando en el conjunto local de datos.

- Iteración 14: Bases de datos del AutoComplete intercambiables.

Se trata de ofrecer la posibilidad de cambiar la fuente de datos asignada a AutoComplete del servidor local a DBpedia (servidor externo) y de DBpedia al servidor local indistintamente, según las necesidades del usuario.

- Iteración 15: Consultas a DBpedia con restricción de tipo.

Se trata de incluir restricciones de tipo para las consultas a DBpedia para mantener la coherencia entre los datos del repositorio local, ya que los recursos obtenidos del conjunto de datos de DBpedia serán especificados como el valor de una propiedad de un recurso almacenado en el repositorio local, y para mantener la consistencia del sistema, es necesario que cumplan las restricciones impuestas por los esquemas y ontologías que estructuran los datos.

- Iteración 16: Refactoring.

Consiste en organizar/estructurar todo el código implementado durante el proyecto en clases y métodos para conseguir un código más claro y escalable.

Desarrollo

Etapa 1

Iteración 1: Formulario genérico de consulta

Introducción

Implementación de un formulario Web para la localización de recursos en un conjunto de datos en RDF.

Objetivo

Empezar a habituarse de forma práctica a los lenguajes estudiados para realizar el proyecto, especialmente a JavaScript y a los formularios HTML, ya que, en principio, es con lo que trabajaremos básicamente.

Desarrollo

Se implementa un formulario simple en HTML con algunas de las propiedades genéricas de los datos contenidos en el conjunto. El conjunto de datos contiene publicaciones, así que las propiedades que se definen como campos del formulario son: *Author, Date, dtStart, dtEnd, Label, Title*. El campo *Date* requiere formato de fecha (dd/mm/aa), y *dtStart* y *dtEnd* funcionan con el formato de fecha más la hora.

```
<div>
   <form id="button-example-form" name="button-example-form" method="post">
        <br>
        <p>Author: </p>
                <p><input type="text"
                        name="http://purl.org/dc/elements/1.1/author">
                </p>

        <fieldset id="dateF">
                <label for="date-field">Date: <p></p></label>
                <input id="date-field" type="text"
                        name="http://purl.org/dc/elements/1.1/date"><br><br>
        </fieldset>

        <fieldset id="dtstartF">
                <label for="dtstart-field">dtstart: <p></p></label>
                <input id="dtstart-field" type= "text"
                        name="http://www.w3.org/2002/12/cal/icaltzd#dtstart"><br><br>
        </fieldset>

        <fieldset id="dtendF">
                <label for="dtend-field">dtend: <p></p></label>
                <input id="dtend-field" type="text"
                        name="http://www.w3.org/2002/12/cal/icaltzd#dtend"><br><br>
        </fieldset>
```

```
        <p>Label: </p>
                <p><input type="text"
                        name="http://www.w3.org/2000/01/rdf-schema#label">
                </p>

        <p>Title: </p>
                <p><input type="text"
                        name="http://purl.org/dc/elements/1.1/title">
                </p>

        <input id="buttonS" type="button"  name="button2" value="Submit"
          onclick="javascript:rhz.sparql(rhizomik.SemanticForms.formToSPARQL(form))">

    </form>
</div>
```

Posteriormente, se le añade un botón para cada entrada con tipo de fecha (*Date*, *dtStart* y *dtEnd*) mediante el cual se accede a un calendario para facilitar la introducción de fechas. Para implementar los calendarios, se utilizan unas librerías de Yahoo! User Interface Library, que a parte de simplificar la implementación del widget, proporcionan un atractivo estilo para los calendarios.

Puesto que los calendarios ya no se usan, no se ha incluido una explicación detallada de su implementación al considerarse irrelevante para el desarrollo del proyecto.

Conclusión

Ha sido un trabajo sencillo pero muy útil, ya que esta primera toma de contacto con HTML y JavaScript ha ayudado a desenvolverse un poco mejor por el entorno, a asimilar algunos conceptos que en la teoría no parecían tan claros, se inculcan los errores más comunes…

Iteración 2: Intérprete SPARQL para el formulario Web

Introducción

Cuando se envía/confirma el formulario, lo que se envía es el conjunto de valores definidos para cada propiedad. Ya que el objetivo del formulario es encontrar, en el conjunto de datos local, los recursos que tengan los valores introducidos para las propiedades indicadas, es necesario lanzar una consulta sobre la base de datos para obtener la respuesta deseada. Por lo tanto, a partir de la información obtenida en el formulario, es decir, de los valores introducidos en los distintos campos del formulario, se debe construir una consulta SPARQL (ya que se trata de conjuntos de datos en RDF) para lanzarla al repositorio a la espera de resultados.

Objetivo

Implementar una función que interprete los campos rellenados de un formulario y los convierta en una consulta SPARQL.

Desarrollo

La función tiene que recorrer todos los campos del formulario para extraer la información de los que tengan algún valor asignado. Dado que un formulario puede tener distintos tipos de entradas, y no todas ellas se pueden leer de la misma forma, se debe distinguir entre las entradas tipo texto y las entradas tipo seleccionable (select).

Se crea una variable donde almacenar las tripletas resultantes y otra para definir las condiciones.

```
formToSPARQL: function (form)
{
        var wheres = "\nWHERE ";
        var filters = "\nFILTER ( ";
        var first = true;
        for (var i = 0; i < form.elements.length; i++)
        {
                if (form.elements[ i ].type === 'text' && form.elements[ i ].value !== "")
                {
                        if (first)
                        {
                                wheres += '{\n';
                                first = false;
                        }
                        else
                        {
                                wheres += '.\n';
                                filters += ' && ' ;
                        }
                                        .
                                        .
                                        .
                }
                else if (form.elements[ i ].type === 'select-one' &&
                  form.elements[ i ].options[form.elements[ i ].selectedIndex].value!== "")
                {
                        if (first)
                        {
                                wheres += '{\n';
                                first = false;
                        }
                        else
                        {
                                wheres += '.\n' ;
                                filters += ' && ';
                        }
                                        .
                                        .
                                        .
                }
        }
                                        .
                                        .
```

```
}
```

Cuando se encuentra una entrada del formulario tipo texto con valor asignado, lo primero que se hace es añadir a la variable correspondiente la tripleta generada a partir de la información disponible. Ésta se obtiene del valor introducido en la entrada, del atributo 'name' de la entrada, que contiene la URI identifidora de la propiedad relacionada, por lo tanto es la relación que hay entre el recurso y el valor, y de una variable que representa al recurso que se está buscando.

Después se comprueba si el contenido de la entrada es una URI, ya que en este caso se ha de especificar una restricción de igualdad encerrando el valor obtenido entre los símbolos menor que y mayor que (< >). En cualquier otro caso, el valor obtenido se ha de encerrar entre comillas (" ").

Además, si no es una URI, se debe comprobar si la cadena introducida contiene caracteres comodín (?, *), ya que en tal caso se deben reemplazar por los caracteres equivalentes para SPARQL. Después se define la condición correspondiente para el valor especificado.

```
var x, y;
wheres += '?r   <' + form.elements[ i ].name + '>   ?x' + i;
if (isURI(form.elements[ i ].value))
{
        filters += '?x' + i + ' = <' + form.elements[ i ].value + '>';
}
else
{
        if (form.elements[ i ].value.indexOf('*' ) >= 0 | |
                                form.elements[ i ].value.indexOf('?') >= 0)
        {
                x = form.elements[ i ].value.replace("*", ".*");
                y = x.replace("?", ".");
                filters += 'regex (?x' + i + ', " '  + y + ' ", "i" )';
        }
        else
        {
                //filters += '?x' + i + ' = "' + form.elements[i].value + '" ';
                filters += 'regex (?x' + i + ', "' + form.elements[ i ].value + '", "i")';
        }
}
```

Para los campos tipo 'select' se hace prácticamente lo mismo, la diferencia es que en lugar de comprobar si se ha introducido algún valor en la entrada, se comprueba si alguna de las opciones proporcionadas por la entrada está seleccionada y qué valor contiene.

```
var selectedValue= form.elements[ i ].options[form.elements[ i ].selectedIndex].value;
wheres += '?r   <' + form.elements[ i ].name + '>   ?x' + i;

if (isURI(selectedValue))
{
```

```
        filters += '?x' + i + ' = <' + selectedValue + '>';
}
else
{
        filters += '?x' + i + ' = "' + selectedValue + '" ';
}
```

Para terminar, se devuelve una cadena con la consulta creada a partir de las tripletas y condiciones obtenidas. El método 'sparql()' del objeto 'rhz' se encargará de codificar la cadena generada y de lanzar la consulta al servidor local.

```
formToSPARQL: function (form)
{
                        .
                        .
                        .

        var query = "DESCRIBE ?r " + wheres + "." + filters + ") \n }";
        return query;
}
```

Conclusión

El conversor funciona correctamente para cualquier formulario HTML sencillo. Igual que en la iteración anterior, esta iteración ha sido especialmente útil para familiarizarse con el lenguaje. Aunque todavía no se puede decir que se haya aprendido a desarrollar el lenguaje correctamente, se gana la confianza y soltura necesarias para realizar pruebas más complejas e ir avanzando en el conocimiento del lenguaje: las alternativas que ofrece, cómo y cuándo utilizarlas…

Iteración 3: Formulario específico de consulta

Introducción

En la esquina derecha superior de la página índice de la interfaz de la aplicación Rhizomer, aparecen varios enlaces que corresponden a las diferentes clases de datos que hay en el repositorio local (People, Project y Publication). La idea es que al presionar cualquiera de estos enlaces, se genere un formulario específico para el tipo de datos seleccionado.

Objetivo

Implementar un formulario que se genere automáticamente en función de un tipo dado de datos. El formulario debe contener como campos todas las propiedades específicas relacionadas con el tipo indicado.

Desarrollo

Primero de todo se debe construir una consulta SPARQL en función del tipo seleccionado. La consulta debe obtener todas las propiedades específicas para el tipo indicado.

```
queryType: function (rhz, type)
{
        var queryPattern = "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
                        PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
                        PREFIX owl: <http://www.w3.org/2002/07/owl#>
                        SELECT ?p
                        WHERE {
                        {?p rdfs:domain ?d. ?t rdfs:subClassOf ?d.
                        FILTER ( ?t = <[types]> && ?d != rdfs:Resource)
                        UNION {?r rdf:type owl:Restriction. ?r owl:onProperty ?p.
                        ?t rdfs:subClassOf ?r. \n FILTER (?t = <[types]>) } }";

        var query = queryPattern.replace("[types]", type).replace("[types]", type);
        rhz.sparqlRDF(query, function(out) {
                                rhizomik.SemanticForms.processQueryType(out, rhz);
                                });
}
```

La respuesta obtenida, es de decir, el conjunto de propiedades específicas del tipo seleccionado, se envía a la función *processQueryType()* para procesar el resultado.

El resultado es un conjunto de datos en RDF, por lo tanto, lo primero que se debe hacer es extraer las soluciones de la estructura RDF.

Para recorrer la estructura RDF se transforma la respuesta obtenida a XML mediante la función *createXMLDocFromText()*. Luego se crea un bucle para extraer el identificador de la propiedad correspondiente de cada resultado.

```
        processQueryType: function (response, rhz)
        {
                var XML = new rhizomik.XMLFactory();
                var xmlResp = XML.createXMLDocFromText(response);
                var nodeArray = xmlResp.getElementsByTagName("value");
                if (nodeArray.length === 0)
                {
                        var nodeArray = xmlResp.getElementsByTagName("rs:value");
                }
                var i = 0;
                var propNameArray = [ ];
                var attributes;
                var resource;

                while(i < nodeArray.length)
                {
                        attributes = nodeArray[ i ].attributes;
                        resource = attributes.getNamedItem("rdf:resource");
```

```
                        propNameArray[ i ] = resource.nodeValue;
                        i++;
                }
                                        .
                                        .
                                        .
}
```

Una vez se dispone de las URLs de las propiedades específicas, se procede a construir el formulario. El formulario tendrá tantos campos como propiedades se han obtenido para el tipo seleccionado, dado que cada entrada del formulario representa una propiedad específica de los recursos con el tipo indicado.

Entonces, se creará una entrada para cada propiedad del conjunto obtenido. Cada campo dispone de una etiqueta identificativa con el nombre/etiqueta de la propiedad, extraído de la URL de la propiedad respectiva; y una entrada tipo texto que almacena el identificador de la propiedad en su atributo *name*.

Cuando se ha terminado de definir el formulario, se muestra en la página de la aplicación mediante el método *showTab()*.

```
processQueryType: function (response, rhz)
{
                        .
                        .
                        .

        var x;
        var label;
        var labelArray = [ ];
        var form = '<div> \n <form id="dynamic-form" name="dynamic-form">\n';
        var textInput = '<input type="text" name ="';
        var textInputS = '';
        var submitButton = '<input id="buttonS" type="button" name="buttonS"
                                value="Submit" onclick = "javascript:
                                rhz.sparql(rhizomik.SemanticForms.formToSPARQL(
                                getElementById(dynamic-form)))"/>\n';

        for (var i = 0; i < propNameArray.length; i++)
        {
                label = "<p  class='Property'> ";
                labelArray[ i ] = propNameArray[ i ].replace("#", "/");
                x = labelArray[ i ].lastIndexOf("/");
                label += labelArray[ i ].slice(x+1);
                textInputS += label + ':' + textInput + propNameArray[ i ] + '"/></p>\n';
        }

        form += textInputS + "<br/>" + submitButton + "</form>\n" + "</div>";
        rhz.showTab("HTML", form);
}
```

Conclusión

Para cada tipo de recurso se genera un formulario específico automáticamente, por lo tanto, se ha cumplido el objetivo propuesto satisfactoriamente.

Iteración 4: Formulario dinámico de consulta

Introducción

La idea es ofrecer la posibilidad de añadir más entradas al formulario, donde cada entrada corresponda a una propiedad genérica, para permitir al usuario realizar una búsqueda más precisa o bien especificar alguna propiedad que no se encuentre en el formulario generado automáticamente.

Objetivo

Implementar un formulario dinámico de manera que el usuario pueda personalizarlo. Una vez generado el formulario específico, permitir agregar nuevas entradas al formulario a través de una caja de texto que funcione con autocompletar. El autocompletado mostrará las propiedades genéricas cuyo nombre coincida en su inicio con la cadena introducida en la caja de texto.

Desarrollo

Hasta ahora, se genera de forma automática un formulario en función de las propiedades específicas de un tipo de datos. Se pretende agregar una entrada suplementaria para añadir propiedades genéricas que el usuario puede especificar. La entrada destinada a la adición de propiedades funcionará con un sistema de auto-completado, de forma que se mostrarán, en función de lo que el usuario haya escrito en la entrada, las distintas propiedades genéricas disponibles en el repositorio local de datos mediante una lista.

Por lo tanto, lo primero que se debe hacer es lanzar una consulta sobre el conjunto local de datos para obtener el conjunto de propiedades genéricas. Para ello, es necesario modificar la función en la que se genera el formulario específico, *processQueryType*. Antes de mostrar el formulario en pantalla, se debe añadir la entrada donde especificar la propiedad que se pretende añadir. También se le agregará al formulario un enlace con el símbolo '+' con la finalidad de confirmar el valor introducido en la entrada para añadir propiedades. Es decir, al presionar el símbolo '+' se añadirá una entrada referente a la propiedad genérica que contenga en ese momento la caja de texto para añadir propiedades.

En resumen, se inserta en el formulario un elemento referencia '+', se define la consulta SPARQL correspondiente, y se lanza la consulta a la base de datos local, enviando la respuesta a la función *dynamicForm()* para que genere la nueva entrada donde añadir propiedades.

```
processQueryType: function (response, rhz)
{
                    .
```

```
                    .
                    .

        var ref = '<a href="#" id="add-property" style="text-decoration:none">+</a>';
        form += textInputS + ref + "<br/>" + submitButton + "</form>\n" + "</div>";

        var query = "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
                    PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
                    PREFIX owl: <http://www.w3.org/2002/07/owl#>
                    SELECT ?p
                    WHERE {  ?p rdf:type ?t.
                    FILTER (?t = rdf:Property || ?t = owl:DatatypeProperty ||
                    ?t=owl:ObjectProperty).
                    OPTIONAL {?p rdfs:domain ?d}.
                    FILTER (?d=rdfs:Resource || ! bound(?d)) }";

        rhz.sparqlRDF(query, function(out) {
                            rhizomik.SemanticForms.dynamicForm(out, rhz, form);
                            });
}
```

Igual que con la propiedades específicas, se extrae de la respuesta recibida en formato RDF el conjunto de resultados.

```
        dynamicForm: function (response, rhz, form)
        {
                var XML = new rhizomik.XMLFactory();
                var xmlResp = XML.createXMLDocFromText(response);
                var nodeArray = xmlResp.getElementsByTagName("value");
                if (nodeArray.length === 0)
                {
                        var nodeArray = xmlResp.getElementsByTagName("rs:value");
                }
                var i = 0;
                var propNameArray = [ ];
                var attributes;
                var resource;

                while(i < nodeArray.length)
                {
                        attributes = nodeArray[ i ].attributes;
                        resource = attributes.getNamedItem("rdf:resource");
                        propNameArray[ i ] = resource.nodeValue;
                        i++;
                }
                                .
                                .
                                .
        }
```

Esta vez, en cambio, no se pretende montar una entrada para cada propiedad, sino que se deben almacenar todas las propiedades en un array, ya que serán el conjunto

de datos con el que funcione la entrada para añadir propiedades, debido a que ésta debe funcionar con autocompletar y mostrar el conjunto de propiedades genéricas en las que coincida el inicio de su etiqueta/label con la cadena escrita.

```
dynamicForm: function (response, rhz, form)
{
                    .
                    .
                    .

        var x;
        var label = [ ];
        var labelArray = [ ];
        var name = [ ];
        for (i = 0; i < propNameArray.length; i++)
        {
                labelArray[ i ] = propNameArray[ i ].replace("#", "/");
                x = labelArray[ i ].lastIndexOf("/");
                label[ i ] = ' "' + labelArray[ i ].slice(x + 1) + '" ';
                name[ i ] = ' "' + propNameArray[ i ] + '" ';
        }
                    .
                    .
                    .
}
```

Cuando se dispone del conjunto de propiedades genéricas almacenadas en un array, se procede a crear la entrada para añadir propiedades. La entrada debe tener una estructura específica para que pueda funcionar con autocompletar: una entrada de texto donde escribe el usuario, un contenedor donde depositar la lista de resultados, y una entrada oculta donde se almacena el valor real del recurso seleccionado.

Además, en el formulario se incluyen dos scripts, uno para dotar la entrada de la función de autocompletar (*autoComplete()*), y el otro, se refiere a una función, *newDialog()*, destinada a lanzar un cuadro de diálogo donde confirmar o cancelar la adición de la propiedad seleccionada.

```
dynamicForm: function (response, rhz, form)
{
                    .
                    .
                    .

        form += '<script type="text/javascript">
                        rhizomik.SemanticForms.newDialog();
                </script>
                <div id="panel-d">
                        <div class="hd">Add a property</div>
                        <div class="bd">
                                <form name="add-newP" id="add-newP">
                                        <div id="myAutoComplete">
                                                <input id="myInput" type="text"><br><br>
                                                <div id="myContainer"></div>
```

```
                                        </div>
                                        <input id="myHidden" type="hidden">
                                    </form>
                                </div>
                            </div>
                            <script type="text/javascript">
                                rhizomik.SemanticForms.autoComplete([ + name + ], [ + label + ]);
                            </script>\';
                            rhz.showTab("HTML", form);
}
```

Conclusión

Para completar el funcionamiento del formulario dinámico se deben ver las siguientes iteraciones.

Iteración 5: AutoComplete

Introducción

Se quiere dotar a la entrada para añadir propiedades del formulario específico de consulta de la funcionalidad de autocompletar.

Objetivo

Implementar mediante widgets y utilidades de YUI Library la función de auto-completado sobre la entrada destinada a añadir propiedades, utilizando el conjunto de propiedades genéricas obtenidas anteriormente como fuente de datos.

Desarrollo

Lo primero que se hace es, a partir de los dos vectores pasados como argumento con el conjunto de etiquetas de las propiedades genéricas y el conjunto de URLs, construir un único array con objetos JSON, de forma que cada objeto represente una propiedad genérica con dos atributos: 'label' y 'url'.

```
    autoComplete: function (name, label)
    {
        var i = 0;
        var genericProp = [ ];
        var genericPropJSON;
        while (i < name.length)
        {
            genericPropJSON = '({label: "' + label[ i ] +
                                '", url: "' + name[ i ] + '"})';
            genericProp[ i ] = eval(genericPropJSON);
            i++;
        }
```

```
                    .
                    .
                    .
    }
```

Después se crea una instancia DataSource de YUI a partir del array de resultados. Esta instancia representa la fuente de datos que será asignada a AutoComplete como base de datos donde debe buscar elementos que coincidan con la cadena introducida en la entrada respectiva. Se debe especificar en la instancia de la fuente de datos los campos que interesan de cada resultado: 'label' y 'url'.

Seguidamente, se crea la instancia de AutoComplete y se especifican algunos de sus atributos de configuración. Para definir la instancia de AutoComplete es necesario especificar una entrada tipo texto mediante su identificador, un contenedor donde depositar la lista de resultados, y una fuente de datos con la que efectuar las comparaciones.

```
autoComplete: function (name, label)
{
                    .
                    .
                    .
    var propertiesDS = new YAHOO.util.LocalDataSource(genericProp);
    propertiesDS.responseSchema = { fields : ["label", "url"] };

    var myAC = new YAHOO.widget.AutoComplete("myInput", "myContainer",
                                                        propertiesDS);
    myAC.resultTypeList = false;
    myAC.typeAhead = true;
    myAC.maxResultsDisplayed = 20;
    myAC.minQueryLength = 2;
    myAC.queryDelay = 0.5;
    myAC.prehighlightClassName = "yui-ac-prehighlight";
    myAC.useShadow = true;
                    .
                    .
                    .
}
```

Para terminar, se define una función 'handler' para tratar con el evento *itemSelectEvent* de AutoComplete, que se produce cuando se selecciona alguno de los elementos de la lista de resultados.

El objeto AutoComplete se encarga de depositar el elemento seleccionado de la lista del contenedor en la entrada de texto, de modo que asigna la cadena resultante como valor de la entrada. Por lo tanto, en este caso, simplemente se asigna la URL de la propiedad seleccionada como valor de la entrada oculta del AutoComplete, ya que el valor de la entrada de texto es la etiqueta de la propiedad seleccionada para que el usuario lo entienda fácilmente, pero el valor real del recurso es su identificador.

```
    autoComplete: function (name, label)
```

```
{
                .
                .
                .

        var myHiddenField = YAHOO.util.Dom.get("myHidden");
        var myHandler = function(sType, aArgs)
        {
                var oData = aArgs[2]; //object literal of selected item's result data
                myHiddenField.value = oData.url;
        };
        myAC.itemSelectEvent.subscribe(myHandler);

        return myAC;
}
```

Conclusión

AutoComplete funciona correctamente y se ha implementado de forma rápida y sencilla gracias a las utilidades proporcionadas por YUI Library.

Iteración 6: Diálogo de confirmación

Introducción

Antes de añadir una propiedad, se considera prudente preguntar al usuario si desea añadir la propiedad seleccionada para evitar malentendidos. Por ejemplo, se puede dar el caso en que el usuario presione el enlace para añadir propiedades sin haber introducido ningún valor previamente en la entrada correspondiente.

Objetivo

Lanzar un cuadro de diálogo confirmativo cuando el usuario seleccione la opción de añadir una propiedad.

Desarrollo

Primero se define un cuadro de diálogo utilizando librerías de YUI Library. Se crea una instancia del objeto Dialog, al cual se le especifican algunos atributos de configuración, entre los que se puede destacar el atributo *buttons*, donde se especifican los botones que se incluirán en el cuadro y se les asignan las funciones que definen el comportamiento de la aplicación tras presionarlos.

El cuadro de diálogo debe aparecer cuando el usuario elija la opción de añadir una propiedad, lo que se hace presionando el símbolo '+' contenido en el formulario. Mediante una utilidad de YUI, se define una escucha para que al presionar el enlace mencionado se muestre el cuadro de diálogo.

```
newDialog: function ()
{
        var addPropDialog = new YAHOO.widget.Dialog("panel-d", {
                                width                   : "450px",
                                fixedcenter             : true,
                                visible                 : false,
                                buttons                 : [ { text      : "Add",
                                                              handler   : handleSubmit },
                                                            { text      : "Cancel",
                                                              handler   : handleCancel }]
                        });

        addPropDialog.render();
        YAHOO.util.Event.addListener("add-property",  "click",  addPropDialog.show,
                                                                addPropDialog,    true);
                                        .
                                        .
                                        .
}
```

Las funciones 'handler' relacionadas con el cuadro de diálogo también se definen dentro de la función *newDialog()*. Cuando se presiona el botón de cancelar, simplemente se oculta el cuadro de diálogo. Pero cuado se confirma la acción, se recupera la etiqueta y la URI identificativa de la propiedad seleccionada, y se llama a la función encargada de agregar la nueva entrada para la propiedad indicada.

```
        newDialog: function ()
        {
                var handleCancel = function()
                {
                        addPropDialog.hide();
                }
                var handleSubmit = function()
                {
                        var lab1 = document.getElementById("myInput");
                        var lab = lab1.value;
                        var url1 = document.getElementById("myHidden");
                        var url = url1.value;
                        rhizomik.SemanticForms.addInput(lab, url);
                        addPropDialog.hide();
                }
                                        .
                                        .
                                        .
        }
```

Al final de la función, se incluye la implementación de un 'tooltip' mediante un widget de YUI Library con el propósito de mostrar un mensaje informativo sobre la función del enlace '+' al pasar con el ratón por encima suyo.

```
newDialog: function ()
{
```

```
                    .
                    .
                    .

            var myTooltip = new YAHOO.widget.Tooltip("tooltip", {
                    context     : "add-property",
                    text        :"Add a new property for the current data search",
                    iframe      : true,
                    showDelay   : 500
            });
}
```

Una vez confirmada la acción de añadir propiedad, se genera la entrada para la propiedad seleccionada. Se crea en función de la etiqueta y el identificador obtenido de la propiedad, y se inserta después de la última entrada que corresponda a una propiedad (al final de las propiedades, pero antes de la entrada para añadir propiedades).

```
addInput: function (label, name)
{
        var newInput = document.createElement("p");
        newInput.innerHTML = label + ': <input type="text" name="' + name + '"/>';
        newInput.className = "Property";
        var inputArray = YAHOO.util.Dom.getElementsByClassName("Property", "p");
        if (inputArray.length > 0)
        {
                YAHOO.util.Dom.insertAfter(newInput,  inputArray[inputArray.length  -1]);
        }
        else
        {
                YAHOO.util.Dom.insertBefore(newInput,
                                        YAHOO.util.Dom.getFirstChild("dynamic-form"));
        }
}
```

Conclusión

El formulario dinámico funciona correctamente, no obstante, el sistema implementado para añadir entradas es un tanto complejo y poco intuitivo, por lo que puede resultar confuso, y por lo tanto puede afectar a la usabilidad de la herramienta.

A continuación se exponen unos diagramas para representar el funcionamiento básico del formulario dinámico implementado a lo largo de estas seis iteraciones. Los diagramas reflejan los métodos JavaScript ejecutados como consecuencia de las principales acciones que puede llevar a cabo el usuario sobre el formulario dinámico, así como el efecto que cada acción produce sobre los elementos HTML del formulario.

El primer diagrama muestra cómo se genera el formulario dinámico de consulta. Los elementos HTML coloreados en gris claro y con el borde discontinuo representan elementos ocultos, es decir, que no son visibles para el usuario hasta que no realice la

acción correspondiente para mostrarlos. Se incluye una breve explicación del proceso de generación del formulario dinámico a continuación del diagrama.

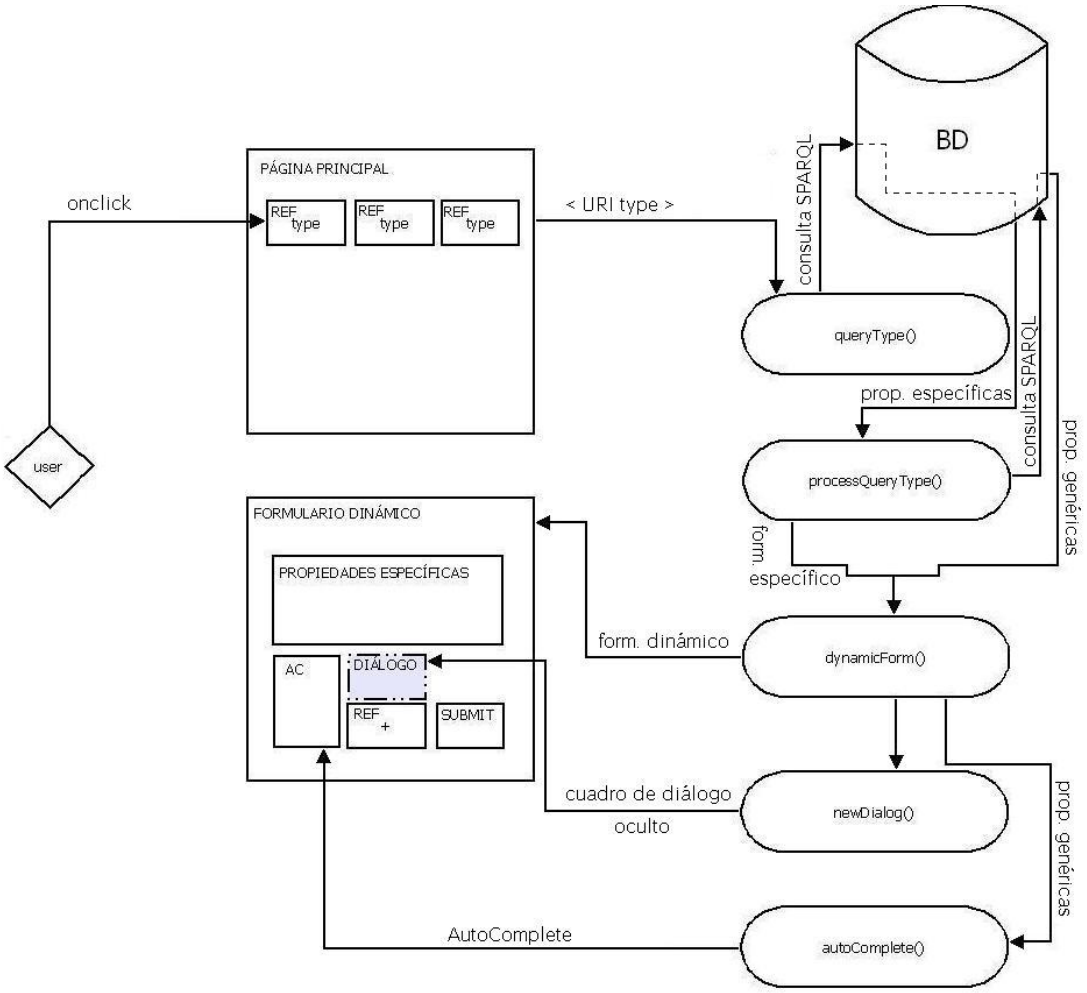

Figura 2: Generación del formulario dinámico de consulta

- Cuando el usuario presiona alguno de los enlaces de tipo contenidos en la página principal de la aplicación, se envía la URL del tipo seleccionado al método queryType().
- Este método lanza una consulta sobre el repositorio local de datos para obtener las propiedades específicas para el tipo indicado, y envía la respuesta al método processQueryType().
- processQueryType() genera un formulario con las propiedades específicas obtenidas y lanza una consulta para obtener las propiedades genéricas. Envía el formulario específico y el conjunto de propiedades genéricas obtenido al método dynamicForm().
- dynamicForm() genera una entrada adicional para añadir propiedades y muestra el formulario.
- También asigna a la entrada para añadir propiedades la función autoComplete() para darle soporte de auto-completado, pasándole el conjunto de propiedades genéricas obtenidas como posibles resultados.
- E inserta un script con la función newDialog() para incluir un cuadro de diálogo de confirmación.

El segundo diagrama muestra los distintos pasos que se deben seguir para añadir una entrada al formulario, entrada que corresponderá a una propiedad. Las distintas acciones que el usuario puede realizar están coloreadas para distinguirlas, y numeradas según el orden requerido para añadir la entrada. A continuación del

diagrama se adjunta una explicación para facilitar el seguimiento y la comprensión del mismo.

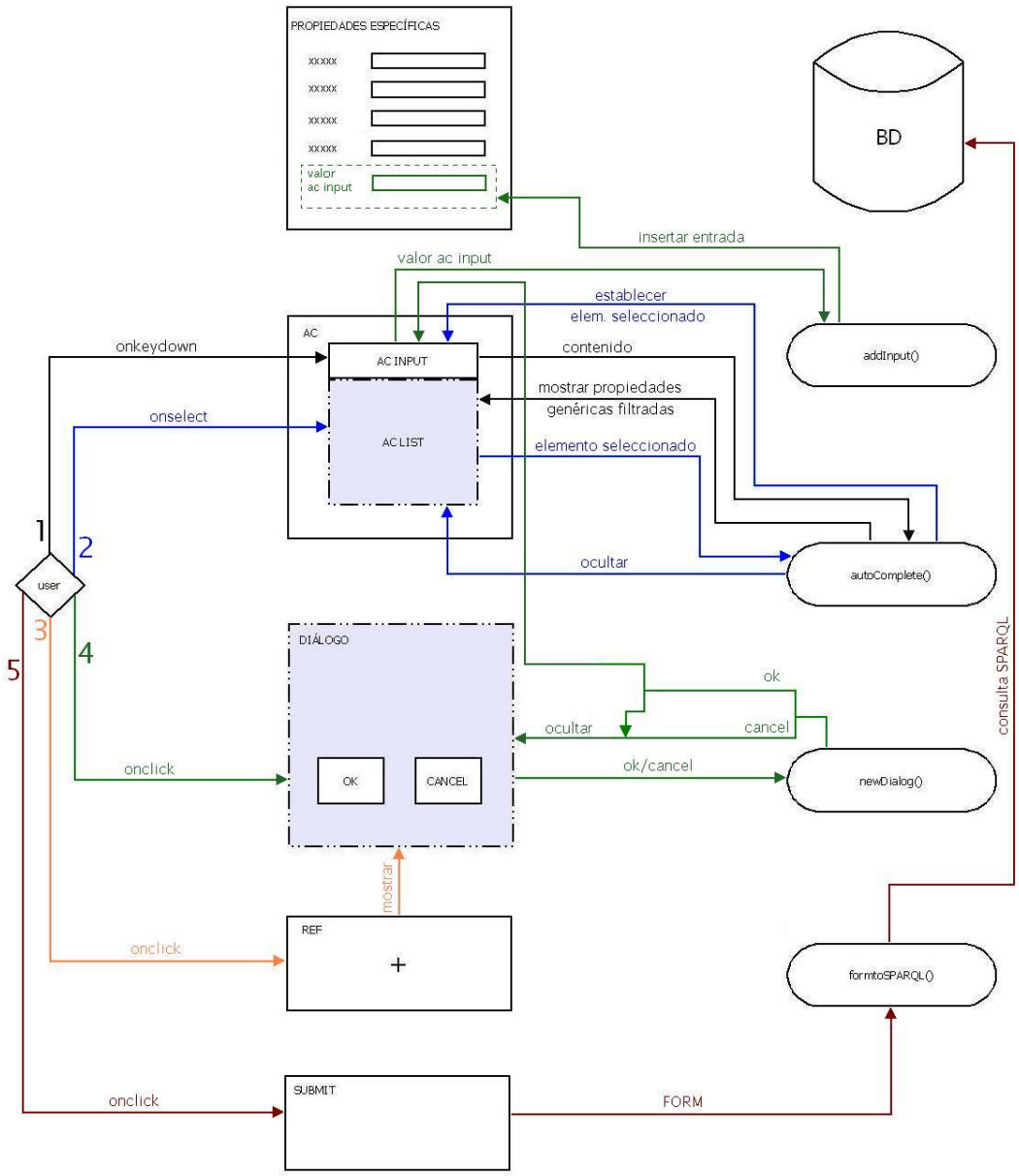

Figura 3: Proceso para añadir una propiedad (Etapa 1)

- [1] Al escribir sobre la entrada de texto con autocompletar, el objeto AutoComplete definido en el método autoComplete() compara la cadena introducida con el conjunto de propiedades genéricas obtenidas durante la generación del formulario, y muestra en el contenedor las que comiencen de igual forma mediante una lista.
- [2] Al seleccionar una de las propiedades mostradas, el objeto AutoComplete generado se encarga de depositar el valor seleccionado en la entrada de texto y de ocultar la lista.
- [3] Al presionar el enlace '+', se muestra el cuadro de diálogo que permanecía oculto hasta el momento.
- [4] El cuadro de diálogo contiene dos botones, uno para confirmar la acción solicitada, y otro para cancelarla. En caso de abortar, simplemente se vuelve a ocultar el diálogo. En caso de confirmar, se obtiene el valor actual de la entrada para añadir propiedades y se llama al método addInput(), que se encarga de insertar una entrada de texto con la propiedad indicada al final del formulario.
- [5] Al presionar el botón 'Submit', se envía el formulario al método formToSPARQL(), que genera una consulta SPARQL a partir de los campos rellenados.

Intervalo entre etapas 1 y 2

Tal y como se ha explicado en la sección de 'Gestión del proyecto', después de la última iteración comentada, se pospuso el desarrollo del proyecto durante un tiempo, tiempo en el que la aplicación sufrió notables cambios.

Para empezar, el formulario dinámico de consulta desaparece para instaurar otros sistemas de localización de recursos. No obstante, se implementan unos formularios de edición de recursos basados en los antiguos formularios de consulta, generados automáticamente en función de las propiedades asignadas al recurso seleccionado, y con la posibilidad de añadir nuevas propiedades mediante una entrada con AutoComplete.

Los formularios de edición se generan a partir de estructuras RDF mediante una transformación XSL. La transformación consiste en recorrer la estructura de uno de los datos contenidos en el repositorio local almacenado en RDF, y a partir de los metadatos contenidos, generar entradas para el formulario correspondientes a las propiedades del recurso seleccionado. La entradas serán de un tipo u otro (literal, XMLSchema, con AutoComplete…) en función del tipo de cada descripción semántica. Además, al lado de cada entrada se inserta una etiqueta identificativa de la propiedad relacionada, enlazada a la descripción semántica de dicha propiedad. A parte de construir una entrada para cada propiedad asignada al recurso, cada transformación XSL genera algunos elementos comunes para el formulario: botones (edit, cancel), enlaces (para añadir propiedades) y un entrada a modo de título del formulario con el identificador del recurso.

Aunque los formularios de edición están basados en el formulario de consulta, poseen numerosas diferencias. Para empezar, se sustituye el complejo sistema para añadir propiedades por otro mucho más simple. Ahora, no hay una entrada permanente en el formulario para añadir propiedades, ni se lanza un cuadro de confirmación antes de añadir la nueva propiedad. Simplemente, el formulario dispone de un enlace con el símbolo '+' en la esquina inferior derecha, justo debajo de la última entrada, que al presionarlo inserta una nueva entrada justo debajo de la última, donde especificar la propiedad que se quiere añadir. Al seleccionar una de las propiedades proporcionadas por la lista de AutoComplete, se habilita inmediatamente una entrada al lado de la propiedad recién definida donde especificar su valor.

Otra diferencia es que la entrada habilitada para depositar el valor de una propiedad recién agregada también funciona con AutoComplete. De la misma forma, todas las entradas que corresponden a los valores de alguna propiedad en el formulario de edición y que sean recursos, funcionan con AutoComplete. Las entradas que contienen valores literales no soportan la función de autocompletar.

Además, el mecanismo de auto-completado para las entradas de recursos funciona en tiempo real. Es decir, para la entrada de propiedades (igual que antes), se lanza una única consulta donde se obtiene el conjunto de datos especificado y se carga en la página. Posteriormente, se filtran del conjunto obtenido los datos en función del valor de la entrada correspondiente. En cambio, para la entrada de recursos, se lanza una consulta sobre la base de datos para cada modificación del valor de la entrada asociada.

Para darle soporte a distintos tipos de entradas de AutoComplete y en distintas situaciones se ha ampliado el método 'autoComplete()' de manera que ahora se utilizan cuatro métodos distintos relativos al AutoComplete para dar la funcionalidad requerida en distintas entradas o circunstancias: 'addProperty()',

‘addPropertyCallback()’, ‘propertyValueAutoComplete()’ y ‘resourceTypeAutoComplete()’.

A continuación se incluyen unos diagramas que representan el funcionamiento de los formularios de edición. El objetivo de los diagramas es reflejar los cambios hechos durante mi ausencia y que se entiendan las diferentes partes y funcionalidades de los nuevos formularios, y los métodos que intervienen y cómo y cúando lo hacen, ya que a partir de la siguiente iteración se trabaja sobre los formularios de edición y se modificarán varios aspectos del formulario.

El siguiente diagrama muestra el proceso a seguir para añadir una propiedad al formulario de edición y asignarle un valor. Representa una relación de las acciones que ha de llevar a cabo el usuario sobre el formulario, las funciones JavaScript que se ejecutan como consecuencia, y el efecto que tiene la ejecución de cada función sobre el formulario.

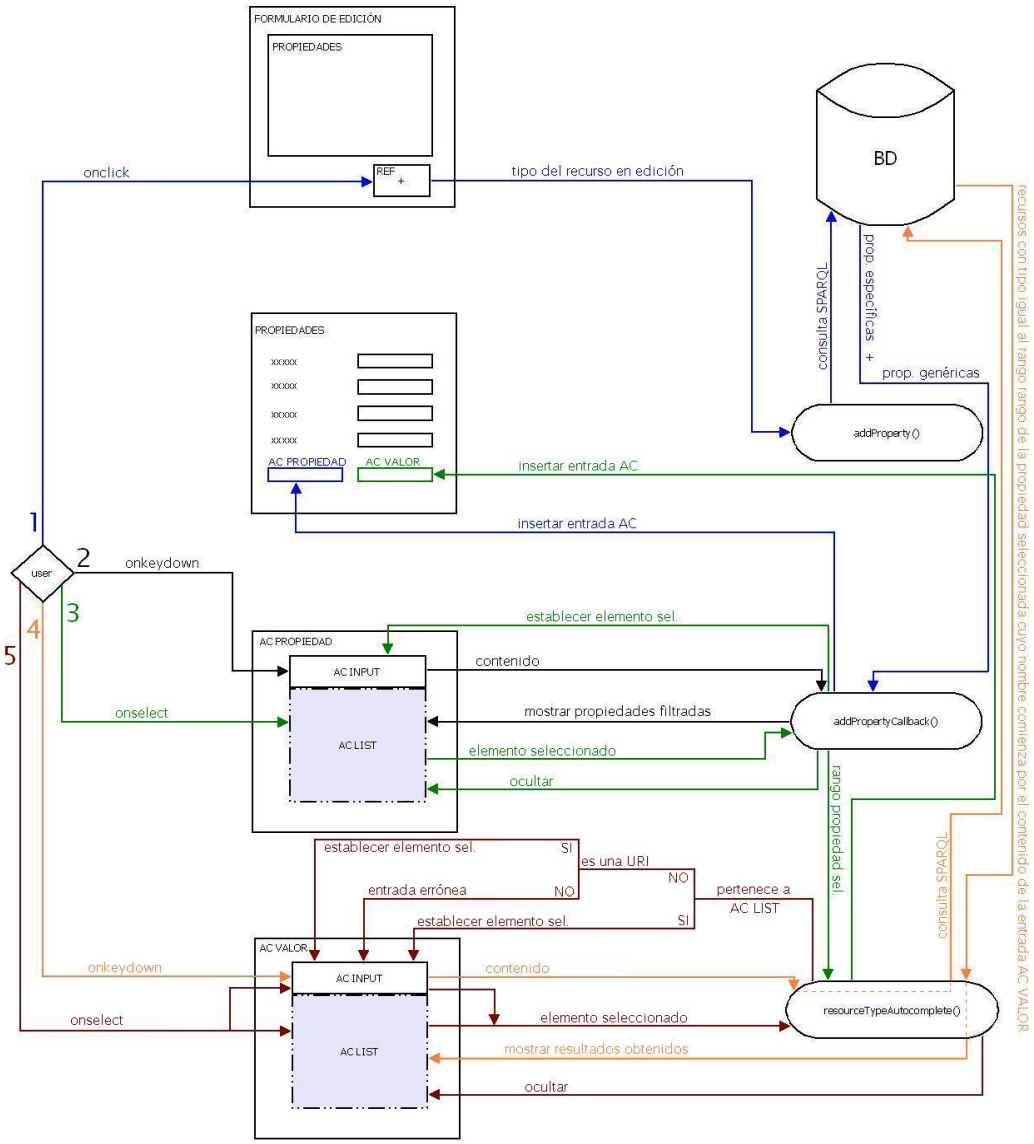

Figura 4: Proceso para añadir una propiedad y asignarle un valor

- [1] Al presionar el enlace ‘+’ del formulario de edición, se envía el tipo del recurso en edición a la función addProperty(), que lanza una consulta SPARQL sobre el conjunto local de datos

para obtener las propiedades relacionadas con el tipo indicado. Éstas se envían a addPropertyCallback() para que genere una entrada con autocompletar con el conjunto de propiedades obtenidas como conjunto de resultados, donde especificar la propiedad que se quiere añadir.

- [2] Al introducir una cadena en la entrada recién creada, se habilita una lista con las propiedades del conjunto obtenido que comiencen de la misma forma.
- [3] Al seleccionar una de las propiedades mostradas, ésta se establece como valor de la entrada de texto y se oculta la lista de resultados. Además, se llama al método resourceTypeAutoComplete(), al que se le proporciona el rango de la propiedad seleccionada, para que genere una entrada de texto donde especificar el valor de la propiedad recién agregada que también funcione con autocompletar.
- [4] Al escribir sobre la entrada para el valor, la función resourceTypeAutoComplete() lanza una consulta sobre el repositorio local de datos para obtener los recursos con el tipo definido por el rango de la propiedad relacionada, o bien alguna subclase suya, cuya etiqueta empiece con la cadena introducida. Al recibir la respuesta, se procesa adecuadamente para mostrar en la lista de resultados el conjunto de recursos obtenidos. Esta vez no se dispone de un conjunto de datos definido, así que para cada modificación del contenido de la entrada de texto, se lanza una consulta al servidor local.
- [5] Cuando se confirma el contenido de la entrada para el valor de la propiedad añadida, se comprueba si se ha seleccionado algún resultado de los proporcionados, y en tal caso se procede a establecer el valor correspondiente en la entrada de texto. En otro caso, si el valor confirmado es una URI, la descripción se toma como válida, pero si no lo es, se marca la entrada como errónea.

El próximo diagrama muestra el comportamiento del formulario de edición cuando se pretende editar una propiedad definida. El proceso es prácticamente el mismo que el que se sigue para asignar un valor a una propiedad recién creada. La diferencia está en que en este caso, se debe obtener el rango de la propiedad a la que pertenece el valor en edición para restringir el tipo de los datos resultantes para el auto-completado. En cambio, en el momento de añadir una propiedad ya se obtiene su rango, de manera que cuando se pretende asignar el valor a la entrada correspondiente ya se dispone del tipo adecuado para realizar la restricción.

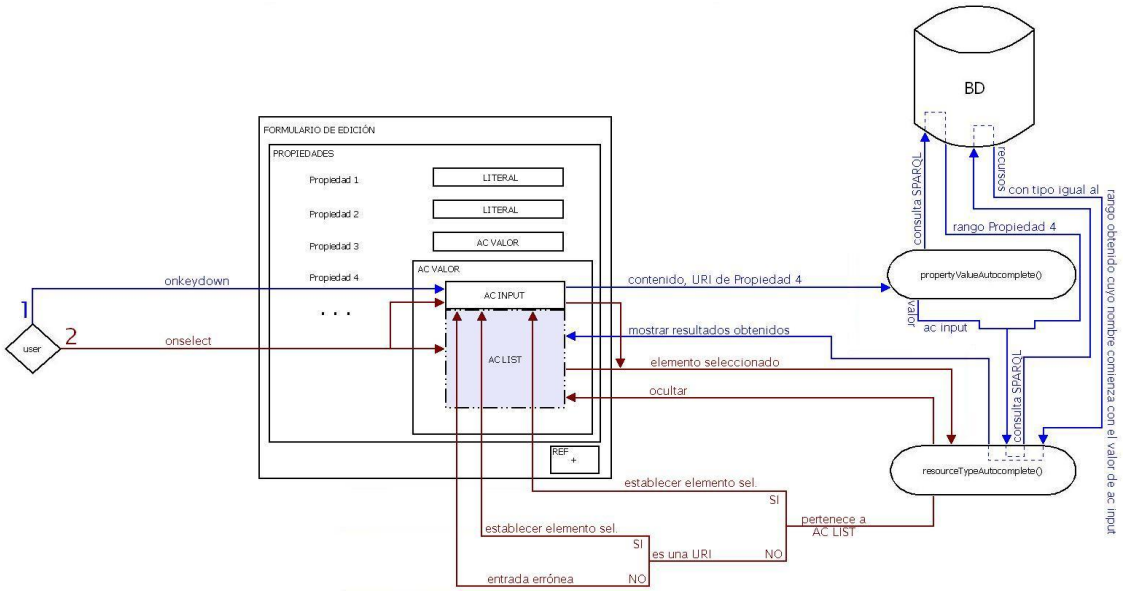

Figura 5: Editar el valor de una propiedad

Es importante dejar claro que de la siguiente iteración en adelante se trabaja sobre el formulario de edición una vez generado. El formulario de edición se genera al seleccionar la opción 'edit' de un recurso concreto, por lo que sirve para editar la descripción semántica de un recurso contenido en el repositorio local de datos.

Etapa 2

A partir de esta iteración, tal y como se ha explicado en la sección anterior, se trabaja sobre el formulario de edición de un recurso específico. El formulario de edición, se genera automáticamente en función de las propiedades definidas para el recurso seleccionado. Además de permitir editar las distintas propiedades definidas, también permite añadir entradas adicionales para especificar más propiedades del recurso.

Iteración 7: Formulario para la definición de descripciones semánticas

Introducción

Cuando se desea modificar el valor para alguna de las propiedades del formulario de edición, la entrada que contiene el valor de la propiedad funciona con una función de autocompletar (siempre y cuando éstas no tengan rango de valores literales), de forma que al introducir una cadena, busca en el repositorio de datos local recursos, que además de cumplir las restricciones impuestas por los esquemas y ontologías definidas (para que un recurso pueda ser el valor de una propiedad, tiene que coincidir su tipo con el rango de la propiedad respectiva), coincida el principio de la etiqueta del recurso con la cadena tecleada. Cuando se añade una propiedad al formulario, se habilita una entrada donde definir el valor para dicha propiedad. La entrada para introducir el valor de la propiedad funciona con el mismo sistema que los campos editables de las propiedades ya definidas. Ya sea para modificar una propiedad o para definirla, siempre que no tenga rango de literal, al introducir una cadena en el campo valor de la propiedad, se muestra una lista con los recursos disponibles cuya etiqueta/nombre comienza por la cadena introducida. Al confirmar el contenido de la entrada en cuestión, se pueden dar tres casos diferenciados:

- Se selecciona uno de los elementos proporcionados por la lista de resultados de AutoComplete: Valor correcto → Propiedad definida/redefinida.
- La cadena introducida es una URI: Valor correcto → Propiedad definida/redefinida.
- La cadena introducida no coincide con ningún elemento de la lista del AutoComplete y no es una URI: Valor incorrecto → Campo del formulario se marca como erróneo.

Objetivo

Implementar un formulario para definir una descripción semántica que se genere de forma automática para el caso en que se introduce un valor 'incorrecto'. De esta forma, cuando el usuario no obtenga el valor que desea de entre los recursos disponibles, puede definir uno a su parecer.

Desarrollo

Se define una función para crear un formulario donde definir la descripción semántica deseada. La función será llamada en el caso en que se introduzca y confirme un valor incorrecto en una entrada editable en el formulario de edición. Admite tres parámetros: 'ac_input', que es la entrada tipo texto donde se está realizando la edición,

'resourceType', que es el tipo del recurso (viene definido por el rango de la propiedad respectiva) representado por una URI, y 'rlabel', que es la etiqueta para referirnos al tipo (se extrae de la URI del tipo).

```
createNewForm: function (ac_input, resourceType, rlabel)
{
                    .
                    .
                    .
}
```

Primero, a partir de la entrada en edición, se obtiene la celda que la contiene.

```
var myInput = ac_input;
var newUri = YAHOO.util.Dom.getAncestorByTagName(myInput, "td");
```

El formulario contendrá en principio tres campos: URI, Label y Type. Todos los campos tendrán un valor por defecto, aunque se pueden modificar en caso necesario.

El campo URI contiene el identificador provisional del nuevo recurso, y se genera en función del tipo y la etiqueta de éste.

El campo Label contiene la cadena introducida por el usuario en el instante en que selecciona la opción de crear una nueva descripción semántica.

En el campo valor de la propiedad Type se establecerá la etiqueta del tipo, ya que el tipo es una URI y queda mucho más claro representar al tipo con el nombre con el que nos referimos a él. En principio, la etiqueta del tipo viene definida por el argumento 'rlabel', pero dependiendo de la circunstancia en que se llama a la función 'createNewForm()', es posible que el tercer parámetro no esté definido.

Por lo tanto, en caso necesario se debe extraer la etiqueta del tipo de la URI que lo define, de la última parte del path. El tipo está almacenado en la variable 'resourceType' y se ha obtenido previamente en una consulta SPARQL comprobando el rango de la propiedad a la que pertenece el campo valor que estamos editando.

```
var myType;
if (rlabel)
{
        myType = rlabel;
}
else if (resourceType)
{
        if (resourceType.indexOf("#") !== -1 )
        {
                myType = resourceType.slice(resourceType.lastIndexOf("#") + 1,
                                                        resourceType.length);
        }
        else if (resourceType.indexOf("/") !== -1 )
        {
                myType = resourceType.slice(resourceType.lastIndexOf("/") + 1,
```

```
                                                        resourceType.length);
                if (myType === "" )
                {
                        myType = resourceType.slice(resourceType.lastIndexOf("/",
                                resourceType.length - 2) + 1, resourceType.length - 1);
                }
        }
}
else
{
        myType = "Resource";
        resourceType = "http://www.w3.org/2000/01/rdf-schema#Resource";
}
```

Seguidamente, se define la entrada para la URI, que será una especie de título del formulario creado. Se creará una URI para el nuevo recurso en función del tipo del recurso y la etiqueta (el nombre que le ha dado el usuario) de esta forma: http://rhizomik.net/[type]/[label].

La entrada para la URI se insertará sobre la celda que contenía la entrada con AutoComplete que originó la generación del formulario, machacando el anterior contenido de dicha celda.

```
        var myUri = "http://rhizomik.net/" + myType.toLowerCase().replace(/ */g, "") +
                        "/" + myInput.value.toLowerCase().replace(/ */g, "");
        newUri.innerHTML = "<input type="text" style="text-align:center"
                        value= + myUri + name="http://www.w3.org/1999/02/22-
                        rdf-syntax-ns#about" >";
```

A continuación, se crea una tabla HTML y se inserta en la celda obtenida. Luego se crean las dos filas necesarias para insertar los campos restantes: Label y Type. Las dos filas estarán contenidas en la tabla. La fila de la URI se compone de una única celda donde se expone la URI del nuevo recurso. En cambio, las demás filas contienen dos celdas, una para la etiqueta, y otra con una caja de texto.

```
        var newTable = document.createElement ("table");
        newUri.appendChild (newTable);

        var newLabel = document.createElement ("tr");
        var newType = document.createElement ("tr");

        var newLabelLabel = document.createElement ("td");
        var newLabelValue = document.createElement ("td");
        var newTypeLabel = document.createElement ("td");
        var newTypeValue = document.createElement ("td");

        newTable.appendChild (newLabel);
        newTable.appendChild (newType);

        newLabel.appendChild (newLabelLabel);
        newLabel.appendChild (newLabelValue);
```

```
newType.appendChild (newTypeLabel);
newType.appendChild (newTypeValue);
```

Las etiquetas de las propiedades estarán enlazadas al elemento de la ontología que las representa, de forma que al clicar sobre alguna de ellas se accede directamente a la descripción semántica que la define.

```
newLabelLabel.innerHTML = " <a class="describe"
        href=" ?query=DESCRIBE%20&lt;http://www.w3.org/2000/01/rdf-
        schema%23label&gt; " onclick=" javascript:rhz.describeResource(
        'http://www.w3.org/2000/01/rdf-schema#label'); return false; "
        title=" Describe http://www.w3.org/2000/01/rdf-schema#label "
        >Label</a> ";

newTypeLabel.innerHTML = "<a class="describe"
        href=" ?query=DESCRIBE%20http://www.w3.org/1999/02/22-rdf-
        syntax-ns%23type " onclick=" javascript:rhz.describeResource(
        'http://www.w3.org/1999/02/22-rdf-syntax-ns#type'); return false;"
        title=" Describe http://www.w3.org/1999/02/22-rdf-syntax-
        ns#type ">type</a>";
```

A la entrada para el valor de Label se le proporciona estructura de literal, con una entrada tipo texto y una entrada tipo 'select' con dos lenguajes (en: English, es: Español), y contendrá lo que haya introducido el usuario en la entrada AutoComplete antes de confirmarla. De todas formas, se puede modificar el contenido en caso necesario.

```
newLabelValue.innerHTML = " <input type="text" value=  + myInput.value +
                name="http://www.w3.org/2000/01/rdf-schema#label"
                class="literal"> <select name='lang'><option></option>
                <option value='en'>en</option>
                <option value='es'>es</option></select>";
```

La entrada para especificar el tipo se ha de construir con la estructura requerida por AutoComplete (una entrada de texto y un contenedor), ya que funcionará con autocompletar. Además, el campo valor de las propiedades que contienen recursos tienen una segunda entrada, pero oculta, que sirve para almacenar el valor real del recurso seleccionado, ya que en la entrada visible para el usuario se le asigna como valor la etiqueta y no la URI, por motivos de claridad. Después de crear la entrada se llama a la función correspondiente para asignarle la funcionalidad de autocompletar.

```
newTypeValue.innerHTML = "<div><input type="text" value=  + myType +
                title=  + resourceType +  class="yui-ac-input"
                autocomplete="off"/>
                <input class="object" type="hidden"
                name="http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
                value= + resourceType + />
```

```
                    <div class="yui-ac-container"></div></div>";

        var newTypeAutoComplete = YAHOO.util.Dom.getFirstChild(newTypeValue);
        rhizomik.SemanticForms.propertyValueAutocomplete(newTypeAutoComplete,
                        resourceType, "http://www.w3.org/1999/02/22-rdf-syntax-ns#type");
```

EL formulario para crear descripciones semánticas también tendrá la opción de agregar más propiedades, igual que el formulario de edición de recursos. De esta forma, el usuario podrá definir más propiedades para el recurso creado a parte de las que contiene el formulario por defecto: URI, Label y Type. Las propiedades que puede añadir se muestran a través de la lista del AutoComplete de la misma forma que en el formulario de edición: se mostrarán solamente las propiedades relacionadas con el tipo del recurso que coincidan (el nombre/etiqueta) con la cadena introducida. A la función encargada de generar la entrada para añadir propiedades, se le ha de pasar el identificador del enlace (que es la URI del recurso) para que sepa donde tiene que insertar la nueva entrada (justo encima del enlace, debajo de la última entrada), y el tipo del recurso que se está definiendo para realizar las restricciones correspondientes en la consulta SPARQL para obtener la lista de propiedades.

```
        var newAddP = document.createElement("tr");
        var newAddProp = document.createElement("td");

        var attr = document.createAttribute("id");
        attr.value = myUri;
        newAddP.setAttributeNode(attr);

        var attr1 = document.createAttribute("colspan");
        attr1.value = "2";
        newAddProp.setAttributeNode(attr1);

        var attr2 = document.createAttribute("style");
        attr2.value = "text-align:left";
        newAddProp.setAttributeNode(attr2);

        newTable.appendChild(newAddP);
        newAddP.appendChild(newAddProp);
        newAddProp.innerHTML = " <a href=
                                "javascript:rhizomik.SemanticForms.addProperty( +
                                 myUri  +, new Array( + resourceType + ))">+</a>";
```

En las siguientes capturas de pantalla se muestra cómo se genera el formulario para definir descripciones semánticas y las principales funcionalidades que ofrece:

- Formulario generado automáticamente que contiene la URI, la etiqueta (label) y el tipo (type) predefinidos del nuevo recurso.
- Añadir una propiedad mediante una entrada de texto asistida con AutoComplete.
- Asignar o editar el valor de una propiedad mediante una entrada de texto asistida con AutoComplete.

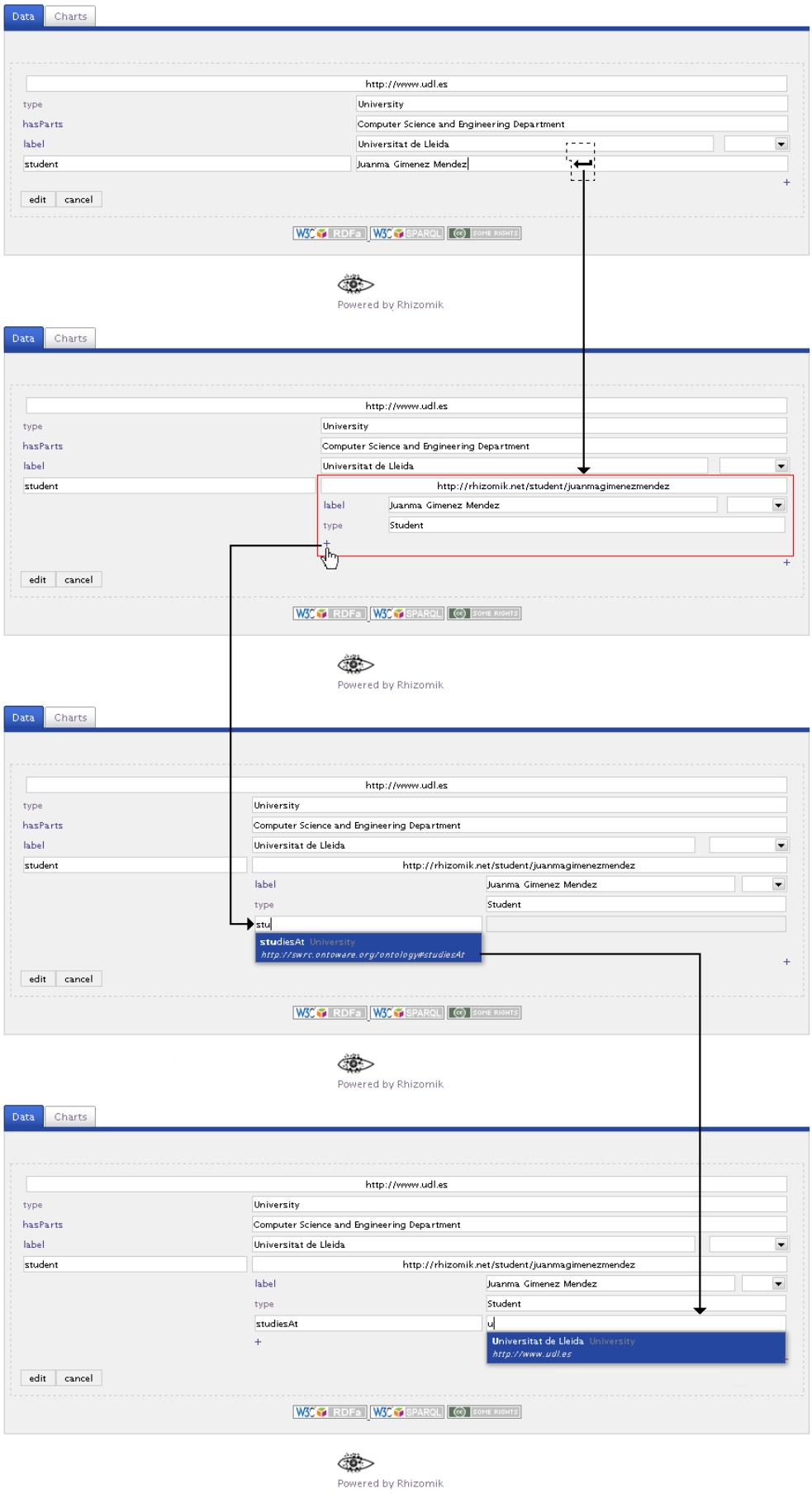

Figura 6: Formulario para la definición de descripciones semánticas

Conclusión

El formulario requerido se genera correctamente, aunque se han de mejorar algunas características.

Iteración 8: Confirmar valor mediante 'enter'

Introducción

El formulario para definir descripciones semánticas se genera en el caso en que se confirme la cadena introducida en el campo valor de una propiedad y no coincida con la etiqueta de ningún recurso que cumpla las restricciones correspondientes. El widget implementado de AutoComplete se encarga de comprobarlo, y mediante el evento 'unmatchedItemSelectEvent' se accede a una función 'handler' para procesar el evento donde se genera el formulario. El evento mencionado implementado por Yahoo no funciona del todo como debiera, ya que al presionar la tecla 'intro', no considera que se esté confirmando el valor introducido, cuando es obvio que la acción de presionar 'intro' debe implicar una confirmación. Este evento sólo se lanza cuando la entrada que contiene el valor no coincidente pierde el foco, ya sea mediante la tecla tabulador o enviándolo a otro lugar con el ratón.

Objetivo

Crear un mecanismo para que en el caso mencionado, también se acceda al formulario para la descripción semántica mediante el 'enter'.

Desarrollo

Se debe realizar una escucha sobre la entrada que se está editando ('elem') para capturar la presión de la tecla 'enter' y realizar las comprobaciones y acciones pertinentes. Para realizar la escucha, se usa una utilidad de YUI Library. Los 'keyCodes' 13 y 9 corresponden a las teclas 'intro' y 'tab'.

```
var enterListener = new YAHOO.util.KeyListener(elem, { keys: [13, 9] },
                                                enterListenerHandler);
enterListener.enable();
```

Al seleccionar uno de los elementos proporcionados por la lista de resultados de AutoComplete, el evento que se encarga de gestionarlo, 'itemSelectEvent', funciona correctamente, de manera que desvía la ejecución a la función indicada para realizar las acciones correspondientes. Por eso, cuando se detecte la presión del 'enter' o el tabulador, es necesario comprobar que el valor confirmado no es algún resultado de AutoComplete, ya que en tal caso el elemento resultante se procesa en otra función.

Para efectuar la comprobación se utiliza un booleano con valor 'false' por defecto que cambiará de estado en el instante en que se seleccione algún resultado de la lista de AutoComplete, de modo que si el valor del booleano es 'true' el resultado ya se trata en otra función.

En caso de que el valor confirmado no coincida con ningún recurso válido, se comprueba si la cadena introducida es una URI. Si lo es, el campo queda correctamente rellenado, de manera que se asigna la nueva descripción semántica a través de su identificador. No obstante, es preciso modificar algunos atributos de la entrada de texto y de la entrada oculta para que hagan referencia al nuevo valor: se establece en el atributo 'value' de la entrada oculta el valor real del recurso (URI), y en el atributo 'title' de la entrada de texto también se establece la URI para que se muestre al pasar por encima de ella.

Si el valor introducido no es una URI, se procede a llamar a la función encargada de generar el formulario donde definir la descripción semántica.

```
var isSelected =  false;
        .
        .
        .

function enterListenerHandler()
{
        var ac_hidden = YAHOO.util.Dom.getNextSibling(ac_input);
        if (!isSelected)
        {
                if (rhizomik.Utils.isURI(ac_input.value))
                {
                        ac_hidden.value = ac_input.value;
                        ac_input.title = ac_input.value;
                }
                else
                {
                        rhizomik.SemanticForms.createNewForm(ac_input,
                                                        resourceType, rlabel);
                }
        }
}
```

A continuación, se define una nueva escucha para el caso en que se ha seleccionado un valor de la lista, y posteriormente se modifica de nuevo la misma entrada. En esta situación, el booleano definido mantiene el valor 'true' que le fue asignado en el momento en que se seleccionó un resultado de la lista, pero no debe ser así, porque el valor ha sido modificado y ya no corresponde al antiguo resultado.

La función indicada se ejecutará cuando cambie el valor de la entrada correspondiente, y lo único que hace es volver a asignar 'false' al booleano para que cuando se confirme de nuevo el valor introducido, se pueda comprobar de nuevo si se trata o no de un elemento seleccionado de la lista del AutoComplete.

```
var isSelectedHandler = function()
{
        isSelected = false;
};
autocomplete.textboxChangeEvent.subscribe(isSelectedHandler);
```

Conclusión

Se ha conseguido habilitar la tecla 'enter' para confirmar un valor introducido en las entradas editables con autocompletar. No obstante, hay situaciones en que no funciona como debería. El sistema para comprobar si la cadena introducida coincide con alguno de los resultados de AutoComplete resulta impreciso cuando se ha modificado varias veces la misma entrada y es necesario buscar una alternativa. La alternativa acordada se expone en la iteración 12.

Iteración 9: De YUI 2 a YUI 3

Introducción

A lo largo del proyecto se utilizan diversas funcionalidades facilitadas por Yahoo! User Interface Library: objetos para la manipulación de elementos DOM, para la instanciación del widget AutoComplete, para la creación de instancias DataSource… Durante el desarrollo del proyecto Yahoo lanza la tercera versión de YUI Library, por lo que se cree conveniente modificar el código para mantenerlo actualizado a la última versión de las librerías que utiliza.

Objetivo

Modificar todo el código relativo a YUI incluido en el archivo 'rhizomer-forms.js' (es el archivo donde se ha trabajado durante el proyecto) para actualizarlo a la última versión.

Desarrollo

La nueva versión de YUI no consiste en una simple reparación de bugs y en añadir algún método u objeto nuevo. YUI 3 modifica por completo el funcionamiento de sus librerías. Cambia la forma de instanciar los objetos, la forma de aplicar métodos o modificar atributos, la forma de suscribirse a eventos, los nombres de los métodos, atributos y eventos…

Un cambio destacable es que antes se tenía que incluir un archivo fuente para cada módulo a utilizar. Cada módulo representa una utilidad o un widget, y contiene un conjunto de clases para la creación y manipulación de utilidades. Con YUI 3 sólo se tiene que incluir un único archivo fuente, pero se tiene que contener todo el código donde se utilicen objetos YUI en un módulo llamado 'sandbox' donde se especificaran todos los módulos incluidos. Por lo tanto, todo el código del archivo 'rhizomer-forms.js' irá encerrado en un 'sandbox' donde se especifican todos los módulos necesarios.

```
YUI().use('autocomplete', 'autocomplete-highlighters', 'autocomplete-filters',
'autocomplete-list', 'node', 'event', 'event-valuechange', 'event-key',
'datasource', 'json', function (Y)
{
        rhizomik.SemanticForms = function()
        {
                .
```

```
                    .
                    .
        }();
});
```

Los principales cambios se realizan sobre objetos para la manipulación de elementos DOM (Node), para la creación de objetos AutoComplete y manipulación mediante sus métodos y atributos (AutoComplete), la instanciación de fuentes de datos que serán asignadas como bases de datos del AutoComplete (DataSource) y la suscripción a eventos (Event). En el bloque de 'Estados del arte', hay una relación de cada uno de estos módulos donde se explica detalladamente para qué sirven y cómo se utilizan, exactamente en la sección de 'Tecnologías relacionadas' en un apartado donde se habla de YUI Library.

Dado que ya se explica ampliamente en otra sección el funcionamiento de los principales objetos de YUI 3 utilizados, no considero necesario comentar todos los cambios realizados, ya que implicaría explicar diez páginas de código que no aportarían prácticamente nada nuevo.

Al realizar el cambio de YUI 2 a YUI 3, se ha aprovechado para efectuar algunas mejoras o modificaciones pendientes, ya que de haberlas hecho antes, se hubieran tenido que volver a modificar porque incluyen código de YUI o bien manipulan objetos YUI.

Por ejemplo, se quería modificar el sistema de consulta para cambiar el formato de respuesta de RDF a JSON, o cambiar la fuente de datos local de AutoComplete por una remota para efectuar las consultas en tiempo real. Estas dos modificaciones se comentan en las siguientes iteraciones.

También se pretendía mejorar un aspecto concreto del formulario de definición de descripciones semánticas. Cuando éste se genera, se crea una URI para la descripción en función de la cadena introducida, que será la etiqueta (Label), y del tipo de recurso. Ambas propiedades se pueden modificar una vez generado el formulario, pero la URI mantiene su valor inicial.

Es necesario implementar una función que actualice el campo URI del formulario cada vez se modifique el valor de alguna de las propiedades predefinidas. Para ello se suscriben las entradas correspondientes a un evento de YUI 3, 'valueChange', que se produce cuando el valor de la entrada asignada cambia.

Entonces, se obtienen las tres entradas implicadas, de los campos URI, Label y Type, y en caso que se produzca el evento suscrito sobre Label o Type, se modificará el valor del campo URI para actualizar su contenido con el valor del campo modificado. Recordemos que la URI generada automáticamente tiene esta forma: http://rhizomik.net/[type]/[label].

```
var uriInput = newUri.one('*');
var labelInput = newLabelValue.one('*');
var typeInput = newTypeAutoComplete.one('*');

var updateUriField = function(e) {

        var updatedLabel = labelInput.get('value');
```

```
        var updatedType = typeInput.get('value');
        uriInput.set('value', "http://rhizomik.net/" +
                        updatedType.toLowerCase().replace(/ */g,"") + "/" +
                        updatedLabel.toLowerCase().replace(/ */g,""));
        };
    labelInput.on('valueChange', updateUriField);
    typeInput.on('valueChange', updateUriField);
```

Conclusión

Aunque ha supuesto más trabajo y más complicaciones de las previstas, se ha realizado la actualización satisfactoriamente. Al implantar un modelo tan diferente para la manipulación de utilidades parecía un tanto complicado, acostumbrado al anterior estilo. Pero al final ha resultado un cambio bastante satisfactorio, porque a parte de actualizar el código a una versión mejorada, una vez comprendido el nuevo sistema, es mucho más simple de utilizar y resulta más modular de manera que queda el código más claro y entendible.

Iteración 10: Trabajar con datos en JSON

Introducción

Cuando la aplicación necesitar realizar una petición sobre el conjunto de datos, ésta lanza una consulta SPARQL a través del método 'sparqlRDF()' del objeto 'rhz'. Este método realiza la consulta indicada en el primer argumento sobre el conjunto de datos local, y devuelve la respuesta en formato RDF a la función 'callback' especificada en el segundo argumento.

Se considera la alternativa de trabajar con datos en formato JSON, ya que es mucho más sencillo procesar la información recibida en formato JSON que en RDF al utilizar la misma notación que los objetos JavaScript.

Objetivo

Realizar las modificaciones necesarias para recibir los conjuntos de resultados de las consultas SPARQL en JSON y procesarlos posteriormente.

Desarrollo

Lo primero es modificar el mecanismo de consulta para cambiar el formato de respuesta de RDF a JSON. Simplemente se ha de sustituir el método utilizado, 'sparqlRDF()', por otro método también definido en el objeto 'rhz' que realiza el mismo procedimiento, pero retorna la respuesta en JSON: sparqlJSON().

Cada vez que se lanza una consulta SPARQL, la respuesta se envía a una función *callback*. La función *callback* necesita una función suplementaria para procesar la respuesta donde se extraen los resultados de la estructura RDF y se depositan en un array. Las funciones encargadas de recorrer la estructura RDF para obtener el conjunto de resultados demandado son *processResults()* y *processSolution()*, después

de haber sido transformada a XML, lo que también requiere la llamada de varios métodos.

Al cambiar el formato de respuesta en las consultas SPARQL a JSON, no es necesario implementar una función suplementaria que se encargue de localizar los resultados, ya que los objetos JSON son fácilmente tratables en JavaScript. No obstante, para el correcto funcionamiento del formulario, se ha de definir una función para procesar los resultados. La razón es que la mayoría de campos del formulario funcionan con AutoComplete, y éste requiere que los datos resultado tengan ciertos campos definidos. Por ejemplo, todos los recursos tienen una URI que los identifica, pero no todos tienen un campo 'Label' donde se define el nombre/etiqueta del recurso. Cuando se selecciona un elemento de la lista proporcionada por AutoComplete, éste coloca en la caja de texto la etiqueta (Label) del recurso, por lo que es necesario que todos los resultados dispongan de un campo 'Label'. Para asignar un atributo 'Label' a los resultados que no tengan, se extrae la última sección del path de la URI que identifica al recurso mediante la función *uriLocalName()*.

Por otro lado, el AutoComplete está configurado para que la lista de resultados se muestre con un formato específico. Por ejemplo, para la entrada de propiedades, a parte de la etiqueta y la URI, se muestra el rango de la propiedad. En realidad, el rango de la propiedad es una URI, de modo que se considera más apropiado mostrar la etiqueta del rango, la cual extraeremos de la URI que lo identifica de la misma forma que en el caso anterior.

Luego se define una función que recibe como argumento la respuesta de una consulta SPARQL en forma de cadena JSON y recorre el objeto JSON para añadir los atributos 'Label' y 'Range Label' a los datos que carezcan de ellos. Para tratar la cadena JSON hay que convertirla en objeto. Se ha utilizado el método *parse()* del objeto JSON de YUI Library dado que antes de realizar la conversión comprueba que se trate de datos, es decir, objetos y vectores; al contrario que el método *eval()* de JavaScript. Después se tiene que volver a convertir el objeto en cadena JSON antes de ser retornado, y se hace mediante el método *stringify()* del objeto JSON.

```
function processJSONResults(jsonStrResults)
{
        var jsonData = Y.JSON.parse(jsonStrResults);
        var resultsArray = jsonData.results.bindings;

        for(var i=0; i<resultsArray.length; i++)
        {
                var result = resultsArray[ i ];
                if (!result.label && result.uri)
                {
                        result.label = {};
                        result.label.type = "literal";
                        result.label.value = rhizomik.Utils.uriLocalname(result.uri.value);
                }
                if (!result.rlabel && result.range)
                {
                        result.rlabel = {};
                        result.rlabel.type = "literal";
                        result.rlabel.value= rhizomik.Utils.uriLocalname(result.range.value);
                }
                jsonData.results.bindings[ i ] = result;
```

```
        }
        jsonStrResults = Y.JSON.stringify(jsonData, null, 3);
        return jsonStrResults;
}
```

También hay múltiples atributos del AutoComplete que requieren ser modificados ya que acceden y tratan con los resultados recibidos, y al tener éstos otro formato, no se pueden procesar de la misma forma. Por ejemplo, en el atributo *resulTextLocator* se indica la localización de la cadena que representa la solución dentro da cada objeto resultado JSON:

```
autocomplete.set("resultTextLocator", "label.value");
```

En el atributo *resultFormatter* se puede especificar un formato personalizado para los elementos de la lista de resultados. Como, en el caso de la entrada para propiedades, por ejemplo, para cada resultado se pretende mostrar la etiqueta, el identificador y el rango, es necesario acceder al atributo *value* de los campos 'label', 'uri' y 'rlabel' de cada objeto JSON:

```
autocomplete.set("resultFormatter", function(query, resultsArray)
        {
                return Y.Array.map(resultsArray, function(result)
                        {
                                var resultTemplate = "<div class='ac-label'>" +
                                                     "{label}" +
                                                     "</div><div class='ac-range'>" +
                                                     "{rangeLabel}" +
                                                     "</div><div class='ac-uri'>" +
                                                     "{uri}" +
                                                     "</div>";
                                var resultJSON = result.raw;
                                return Y.Lang.sub(resultTemplate, {
                                                     label      : resultJSON.label.value,
                                                     rangeLabel : resultJSON.rlabel.value,
                                                     uri        : resultJSON.uri.value
                                                  });
                        });
        });
```

En la función encargada de procesar el evento *select*, que se produce al seleccionar algún elemento de la lista de resultados de AutoComplete, también se accede a algunos campos del resultado seleccionado.

El evento *select* proporciona en el argumento 'e' un atributo *result* que contiene el objeto resultado seleccionado. Cada objeto resultado de YUI 3 se compone de 4 atributos:

- text: Cadena de texto que será colocada en la entrada de texto en caso de ser seleccionado este resultado (se define su localización en el atributo *resultTextLocator*).

- display: Resultado formateado para mostrar al usuario en la lista del contenedor.

- highlighted: Cadena de texto resultado (result.text) remarcada. Sólo está disponible cuando se ha activado algún filtro de remarcado (resultHighlighters).

- raw: Resultado en el formato proporcionado por la base de datos.

Como se necesita acceder a varios campos del resultado, se empleará la propiedad 'raw' del resultado dado que es el objeto JSON enviado por la fuente de datos.

Al seleccionar un elemento, AutoComplete se encarga de depositar el resultado en la caja de texto. Pero todavía hace falta modificar algunos atributos de las entradas con los valores correspondientes. Por ejemplo, para la entrada del valor de las propiedades, es decir, la entrada de los recursos, se le asigna la URI del elemento seleccionado como *title* a la entrada de texto. De esta forma, se añade un tooltip informativo del identificador del recurso. A la entrada oculta se le debe de asignar como valor el identificador del recurso, ya que la entrada de texto contiene como valor la etiqueta.

```
var isSelected = false;
var itemSelectEventHandler = function(e) {

                    var result = e.result.raw;
                    var ac_hidden = ac_input.next();
                    isSelected = true;
                    ac_input.set('title', result.uri.value);
                    ac_hidden.set('value', result.uri.value);
                    };
autocomplete.on("select", itemSelectEventHandler);
```

Conclusión

Con el cambio realizado es mucho más sencillo procesar los resultados, y en consecuencia, se pueden elaborar métodos para el procesamiento de resultados más genéricos.

Iteración 11: AutoComplete en tiempo real

Introducción

Cuando se introduce una cadena en una entrada del formulario que funciona con AutoComplete, éste realiza una consulta sobre la fuente de datos asignada en busca de datos coincidentes. La fuente de datos asignada al AutoComplete referente a la entrada de las propiedades, es una fuente de datos local, de manera que cuando se efectúa la opción de 'añadir propiedad', se lanza una consulta para obtener el conjunto de propiedades contenidas en el servidor local que cumplan con las restricciones impuestas (las propiedades deben estar relacionadas con el tipo del recurso seleccionado) y se almacenan en una variable en forma de vector.

Entonces, cada vez que AutoComplete busca elementos coincidentes con la cadena introducida en la caja de texto correspondiente, lo hace sobre el vector donde se han almacenado las propiedades.

Objetivo

Se pretende cambiar la fuente de datos con la que trabaja AutoComplete para que realice cada consulta directamente sobre el servidor local. De esta forma no es necesario cargar todo el conjunto de datos en la página, que puede llegar a ser muy amplio.

Desarrollo

FUENTE DE DATOS LOCAL:

La fuente de datos local se crea instanciando un objeto 'Y.DataSource.Local' de YUI, donde se especifica el conjunto de datos mediante un objeto o un vector JavaScript. En nuestro caso, le pasamos un objeto JSON que contiene un vector con objetos resultado.

Se obtiene el conjunto de propiedades específicas que cumple los requisitos para el recurso en edición mediante una consulta SPARQL, y la respuesta se trata en el método *processJSONResults()*, que añade los campos 'Label' y 'Range Label' a los resultados que no dispongan de ellos. Las propiedades genéricas se han obtenido anteriormente (en el momento de generar el formulario de edición) de la misma forma, mediante una consulta SPARQL, la respuesta de la cual ya ha sido procesada por *processJSONResults()* y almacenda en el atributo *genericProperties*.

Así que se juntan la propiedades específicas para el tipo de recurso con las propiedades genéricas (válidas para cualquier tipo) en un solo objeto JSON mediante la función *joinProperties()*, y el objeto resultante será asignado como conjunto de datos de la fuente de datos local, la cual será asignada como fuente de datos del AutoComplete.

Después se añade a la fuente de datos local un *plugin* donde especificar el 'schema' para la estructura de datos JSON. Se le indica donde se encuentra el vector de resultados, y qué campos contiene cada objeto resultado.

```
var especificProperties = processJSONResults(response);
var properties = joinProperties(especificProperties, genericProperties);

var propertiesDS = new Y.DataSource.Local({source: properties});
propertiesDS.plug({fn: Y.Plugin.DataSourceJSONSchema, cfg: {
                schema: {
                        resultListLocator       : "results.bindings",
                        resultFields            : ["label", "uri", "range", "rlabel"]
                }
        }});
```

FUENTE DE DATOS REMOTA:

La fuente de datos remota se crea instanciando un objeto 'Y.DataSource.IO' donde se especifica la ubicación del conjunto de datos mediante una URL. La URL proporcionada a la fuente de datos remota se obtiene del servidor local.

Se debe especificar la cabecera de las peticiones al servidor para indicar que los datos resultantes deben ser en formato JSON.

También en este caso se debe especificar el 'schema' de la estructura de datos JSON para localizar el vector de resultados e indicar los campos requeridos para cada dato.

```
var properties = rhz.getBaseURL();
var propertiesDS = new Y.DataSource.IO({source: properties});

Y.io.header('accept', 'application/json');
propertiesDS.plug({fn: Y.Plugin.DataSourceJSONSchema, cfg: {
        schema: {
            resultListLocator  : "results.bindings",
            resultFields       : ["label", "uri", "range", "rlabel"]
        }
    }});
```

AutoComplete se encarga de buscar en la fuente de datos indicada recursos cuyo nombre coincida (en cómo coincida depende de los parámetros de configuración especificados en el objeto AutoComplete) con la cadena introducida en la entrada asociada, pero no entiende de tipos y rangos.

Cuando se trabajaba con la fuente de datos local, se lanzaba una única consulta donde se obtenían todas las propiedades ya filtradas, es decir, válidas para el tipo de recurso que se está editando. El conjunto de propiedades filtradas por dominio era asignado al AutoComplete como fuente de datos, de forma que sólo tenía que comprobar la coincidencia de caracteres en cada consulta. En cambio, con la fuente de datos remota, no se dispone de un conjunto de datos predefinido, sino que para cada consulta que realice el AutoComplete tiene que acceder al servidor y buscar entre todo el repositorio de datos. Luego es necesario modificar el patrón de consulta del AutoComplete para que además de comprobar la cadena de la etiqueta, compruebe que los datos resultantes sean propiedades, y de la compatibilidad de cada propiedad para el tipo correspondiente de recurso.

```
autocomplete.set("requestTemplate", function(sQuery)
    {
        var queryPattern =
            //Propiedades específicas para el tipo 'resourceTypes[0]'
            "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
            PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
            PREFIX owl: <http://www.w3.org/2002/07/owl#>
            SELECT DISTINCT ?uri ?label ?range ?rlabel WHERE {
            { ?uri rdfs:domain ?d. <[types]> rdfs:subClassOf ?d.
            OPTIONAL { ?uri rdfs:label ?label }
            OPTIONAL { ?uri rdfs:range ?range }
            OPTIONAL { ?range rdfs:label ?rlabel }
```

```
                    FILTER (?d != rdfs:Resource) }
                    UNION { ?r rdf:type owl:Restriction; owl:onProperty ?uri. <[types]>
                                                        rdfs:subClassOf ?r.
                    OPTIONAL { ?uri rdfs:label ?label }
                    OPTIONAL { ?r owl:allValuesFrom ?range }
                    OPTIONAL { ?r owl:someValuesFrom ?range }
                    OPTIONAL { ?range rdfs:label ?rlabel } }" +
                    //Propiedades genéricas
                    "UNION { ?uri rdf:type ?t.
                    OPTIONAL { ?uri rdfs:label ?label }
                    OPTIONAL { ?uri rdfs:domain ?d }
                    OPTIONAL { ?uri rdfs:range ?range }
                    OPTIONAL { ?range rdfs:label ?rlabel }
                    FILTER ( (?d=rdfs:Resource || !bound(?d)) &&
                            (?t = rdf:Property || ?t = owl:DatatypeProperty ||
                            ?t=owl:ObjectProperty || ?t=owl:AnnotationProperty) )}}
                    LIMIT 500";

            var query = queryPattern.replace(/\[types\]/g, resourceTypes[ 0 ]);
            return "?query=" + encodeURIComponent(query);
    });
```

Debido a que ahora no se dispone de un conjunto de resultados predefinido, no se puede aplicar la función *processJSONResults()* para procesar adecuadamente todo el conjunto de datos de sola una vez. En su lugar, se utilizará el atributo *resultTextLocator* de AutoComplete para realizar las mismas acciones.

En realidad, este atributo sirve para especificar la ruta dentro del objeto resultado donde se encuentra la cadena que será depositada en la entrada correspondiente al seleccionar dicho resultado. De hecho, antes de realizar el cambio de fuente de datos, el valor del atributo mencionado era una cadena que indicaba dónde localizar el texto que representa al resultado. Pero como el valor de este atributo puede ser una cadena o una función, se definirá como una función para realizar las modificaciones necesarias tal y como se hacía en *processJSONResults()*, ya que AutoComplete aplica la función indicada a cada resultado recibido de la fuente de datos.

```
    //Con fuente de datos local

autocomplete.set("resultTextLocator", "label.value");

    //Con fuente de datos remota

autocomplete.set("resultTextLocator", function(result)
    {
        if (!result.label && result.uri)
        {
            result.label = {};
            result.label.type = "literal";
            result.label.value = rhizomik.Utils.uriLocalname(result.uri.value);
        }
        if (!result.rlabel && result.range)
        {
```

```
                result.rlabel = {};
                result.rlabel.type = "literal";
                result.rlabel.value=rhizomik.Utils.uriLocalname(result.range.value);
            }
            return result.label.value;
        });
```

Conclusión

La fuente de datos asignada al AutoComplete para las propiedades funciona correctamente. Ahora todas las consultas de AutoComplete se realizan en tiempo real.

Iteración 12: Nueva descripción semántica como primer resultado de AutoComplete

Introducción

En la octava iteración se explica como se implementa un sistema para poder confirmar el valor editado de una propiedad que no coincida con ningún elemento proporcionado por la lista del AutoComplete mediante la tecla 'enter'. Cuando se confirma un valor no coincidente, significa que el recurso que se pretende definir como valor de una propiedad no existe en la base de datos local. Entonces, se genera un formulario donde definir algunas propiedades para el nuevo recurso. A simple vista funciona bien, pero cuando se modifica y confirma varias veces el contenido de la misma entrada presionando la tecla 'enter', se obtiene un comportamiento inesperado.

Objetivo

Configurar el AutoComplete para que muestre como primer elemento de la lista de resultados la opción de crear un nuevo recurso, recurso que estará destinado a la descripción semántica de otro recurso ya que será el valor de una de sus propiedades. Al seleccionar la opción de 'New Resource', se debe generar el formulario para la definición del recurso, igual que hasta ahora. De esta forma siempre se tendrá que escoger uno de los resultados de la lista del AutoComplete para que la entrada sea válida, a no ser que sea una URI y exceptuando las entradas que contienen valores literales, ya que éstas no funcionan con AutoComplete.

Desarrollo

Se pretende añadir un objeto resultado al conjunto de resultados con el que trabaja AutoComplete. Por un lado, el objeto resultado tiene que tener una estructura concreta para que AutoComplete sea capaz de interpretarlo. Debe tener un atributo 'raw', donde se almacena el resultado en el formato proporcionado por la fuente de datos (JSON), un atributo 'text', que contiene el texto que se colocará en la entrada al seleccionar el resultado, y un atributo 'display', que contiene el formato en que se mostrará el resultado dentro de la lista.

Por otro lado, el objeto debe contener la información que se desea mostrar al usuario. En este caso, se pretende informar de que se puede crear un nuevo recurso a partir de

la cadena introducida. También se muestra una URI provisional para el recurso, generada a partir de la etiqueta (lo que ha escrito el usuario) y el tipo del recurso, definido por el rango de la propiedad a la que pertenece, cuya etiqueta está almacenada en la variable 'rlabel'.

Para acceder al conjunto de resultados que recibe AutoComplete y posteriormente muestra en la lista, se emplea el evento *results* asociado al objeto AutoComplete, que desvía la ejecución hacia la función indicada en la suscripción en el momento en que el AutoComplete recibe un conjunto de resultados de la fuente de datos, aunque sea vacío.

En la función indicada se crea un objeto con la estructura requerida por AutoComplete y el contenido deseado, y posteriormente se inserta en el primer lugar del vector de resultados para que se muestre el primero de la lista, ya que AutoComplete muestra los resultados en el mismo orden en el que están almacenados en el array.

```
var responseHandler = function(e)
{
        var newResource = {};
        var tempUri = "http://rhizomik.net/" + rlabel.toLowerCase().replace(/ */g, "") +
                      "/" + ac_input.get('value').toLowerCase().replace(/ */g, "");
        newResource.raw = { "uri" : { "type" : "uri", "value" : tempUri },
                            "label" : { "type" : "literal", "value" : ac_input.get('value')}
                          };

        var newRTemplate = "<div class='ac-label'>" +
                           "{label}" +
                           "</div><div class='ac-range'>" +
                           "New Resource" +
                           "</div><div class='ac-uri'>" +
                           "{uri}" +
                           "</div>";
        var newRDisplay = Y.Lang.sub(newRTemplate,{
                                label   : ac_input.get('value'),
                                uri     : tempUri
                                });
        newResource.display = newRDisplay;
        newResource.text = ac_input.get('value');
        e.results.unshift(newResource);
};
autocomplete.on("results", responseHandler);
```

Ahora se dispone de un objeto resultado informativo para crear una descripción semántica, pero para que seleccionarlo implique la generación del formulario para su definición, es necesario asignarle tal comportamiento.

Para ello, se utiliza el evento *select* del objeto AutoComplete, que detiene la ejecución al seleccionar uno de los elementos de la lista de resultados y permite la manipulación del elemento seleccionado para procesar el resultado. Aquí se comprueba si el elemento seleccionado es el primero de la lista ('New Resource'), y en tal caso que no se trate de una URI. Si se dan las dos circunstancias, se llama a la función encargada de generar el formulario para la descripción semántica.

```
var itemSelectEventHandler = function(e)
{
        var aux = e.result.display;
                .
                .
                .

        if(aux.indexOf("New Resource") !== -1)
        {
                if (rhizomik.Utils.isURI(ac_input.get('value'))
                {
                        .
                        .
                        .
                }
                else
                {
                        rhizomik.SemanticForms.createNewForm(ac_input,
                                                        resourceType, rlabel);
                }
        }
        else
        {
                .
                .
                .
        }
};
autocomplete.on("select", itemSelectEventHandler);
```

Conclusión

La solución ideada se ha implementado correctamente. Además de funcionar perfectamente, sin los anteriores errores, resulta práctica y elegante, y sobre todo más intuitiva para el usuario la forma de crear la descripción.

En las siguientes capturas de pantalla, se puede observar el modo en que se muestra la opción de definir un nuevo recurso en las ontologías utilizadas durante el proceso de edición del valor de alguna propiedad de un recurso existente. Al seleccionar el elemento adecuado de la lista de resultados de AutoComplete, se genera el formulario donde especificar algunas propiedades en caso necesario.

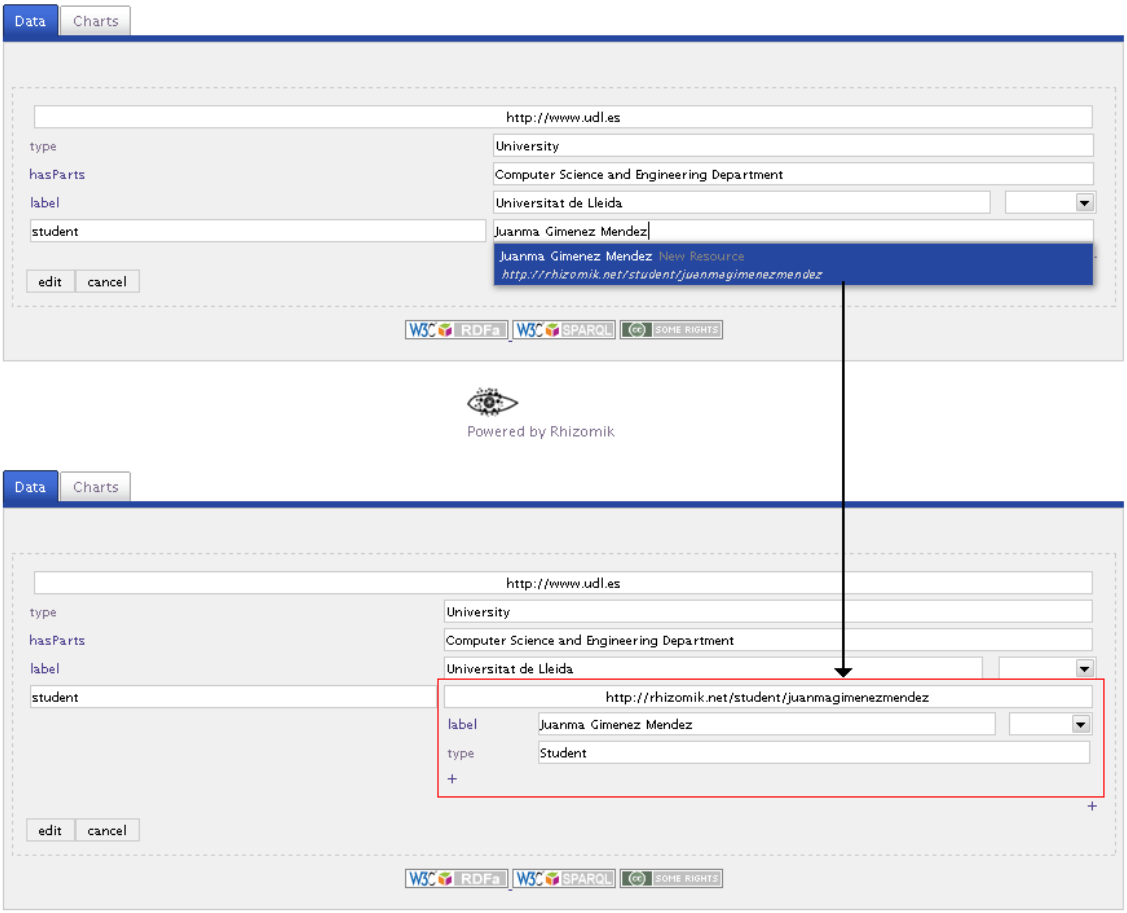

Figura 7: Nueva descripción semántica como primer resultado de AutoComplete

Iteración 13: Consultas de AutoComplete en fuentes de datos externas

Introducción

Se considera la alternativa de ofrecer la posibilidad de realizar las consultas del AutoComplete sobre una fuente de datos externa, almacenada en un servidor ubicado en un dominio distinto al que pertenece la aplicación. De esta forma, si AutoComplete, durante la edición de la propiedad de un recurso, no proporciona al usuario ningún resultado para la cadena introducida, puede realizar la consulta en otra base de datos, y si encuentra allí lo que necesita, se ahorra el proceso de definir la descripción semántica.

Objetivo

Configurar el objeto AutoComplete para permitir cambiar la fuente de datos de forma que se puedan realizar consultas a servidores externos, como por ejemplo, DBpedia. La opción de cambiar de base de datos se mostrará como segundo elemento de la lista de resultados del AutoComplete.

Desarrollo

Primero de todo se procede a crear una instancia de DataSource. Esta vez, como se pretende acceder a un dominio distinto del que pertenece la aplicación en busca de datos, se empleará una instancia DataSource.Get en lugar de DataSource.IO, ya que ésta última sólo permite la comunicación con un servidor ubicado en el mismo dominio que la aplicación.

En la definición de la instancia de DataSource, se debe especificar la URL a través de la cual se accede a la base de datos correspondiente. Se ha decidido utilizar DBpedia como fuente de datos alternativa ya que contiene un gran y variado conjunto de datos semánticos. Se accederá al conjunto de datos http://dbpedia.org, disponible en http://omediadis.udl.cat:8890, un repositorio de datos basado en Virtuoso. Como el repositorio local de datos también está basado en Virtuoso, los conjuntos de resultados recibidos de http://omediadis.udl.cat:8890 tendrán el mismo formato (JSON) y exactamente la misma estructura que los datos contenidos en el servidor local con los que se acostumbra a trabajar. Entonces, no hace falta someter a los datos recibidos a ningún tipo de procesamiento especial, sino que serán tratados de la misma forma que los datos provinentes del servidor local.

En principio se iba a utilizar http://lod.openlinksw.com como repositorio para acceder al conjunto de datos de DBpedia, y aunque hay que decir que funcionaba bastante más rápido que http://omediadis.udl.cat:8890, según parece, están actualizando la herramienta, y en numerosas ocasiones no se encuentra disponible. Sólo es cuestión de esperar a que terminen, no obstante, mientras tanto, es preferible utilizar una plataforma menos veloz pero con más garantía de disponibilidad.

Seguidamente de la URL de omediadis.udl.cat, especificaremos un parámetro para el lenguaje de consulta (SPARQL), y un segundo parámetro donde se indica la ubicación de la base de datos sobre la que se quiere consultar (http://dbpedia.org).

Es necesario definir un 'schema' para la fuente de datos donde indicarle la localización del vector de resultados dentro de cada objeto JSON, y los campos relevantes de cada resultado. Esta instancia de AutoComplete está asignada a los campos valor de las propiedades que contienen recursos, de modo que los campos que interesan de cada resultado son 'Label', 'URI' y 'Type'.

```
        var dbpediaURL =
"http://omediadis.udl.cat:8890/sparql?default-graph-uri=http%3A%2F%2Fdbpedia.org";
        var dbpediaDS = new Y.DataSource.Get({ source : dbpediaURL });
        dbpediaDS.plug({fn: Y.Plugin.DataSourceJSONSchema, cfg: {
                                schema: {
                                        resultListLocator       : "results.bindings",
                                        resultFields            : [ "label", "uri", "type" ]
                                        }
                        }});
```

Después se tiene que crear un objeto resultado para insertarlo como segundo elemento en el vector que contiene el conjunto de resultados de AutoComplete. Este resultado informará al usuario de la posibilidad de realizar la búsqueda en DBpedia. El objeto resultado se crea e inserta en el array de resultados de AutoComplete de la misma forma en que se hace con el resultado para definir un nuevo recurso, explicado en la iteración anterior. Este proceso se realiza en la función *responseHandler()*, que

se ejecuta cuando se produce el evento *results*, es decir, cuando el objeto AutoComplete recibe un conjunto de resultados, aunque sea vacío.

```
var responseHandler = function(e)
{
        /* DEFINICIÓN newResource */
                        .
                        .
                        .

        var dbpediaLookUp = { };
        dbpediaLookUp.raw = {"label" : {"type" : "literal",
                                        "value" : ac_input.get('value')}};
        dbpediaLookUp.text = ac_input.get('value');
        dbpediaLookUp.display =     "<div class='ac-label'>" +
                                    ac_input.get('value') +
                                    "</div><div class='ac-range'>" +
                                    "Look up in DBpedia" +
                                    "</div><div class='ac-uri'>" +
                                    "http://dbpedia.org" +
                                    "</div>";

        e.results.unshift(dbpediaLookUp);
        e.results.unshift(newResource);
};
autocomplete.on("results", responseHandler);
```

Ahora se tiene que implementar el comportamiento que debe realizar AutoComplete cuando se seleccione el resultado recién definido. Como se quiere realizar la consulta en otra base de datos, se tiene que cambiar la fuente de datos asignada al AutoComplete por la fuente de dates definida recientemente en una instancia de DataSource.Get.

Cabe recordar que el atributo *requestTemplate* del objeto AutoComplete contiene una función que devuelve una cadena con la consulta SPARQL correspondiente. Durante la definición del objeto AutoComplete, se le asigna la función *localQuery()* al atributo *requestTemplate*, que devuelve una consulta SPARQL codificada para localizar los recursos dentro del conjunto de datos local con el mismo tipo que el rango definido de la propiedad respectiva, o bien alguna subclase suya; y el principio del nombre/etiqueta (Label) del recurso igual a la cadena introducida. Dado que la consulta a realizar sobre DBpedia es ligeramente distinta, se crea una función *dbpQuery()* donde se define la consulta SPARQL a realizar sobre DBpedia en función de la cadena introducida. Esta función se analizará más adelante.

AutoComplete, para cada modificación del valor de la entrada de texto asignada, ya sea introduciendo un carácter o bien borrando, lanza una consulta sobre el conjunto de datos correspondiente. Cuando el usuario decide buscar en DBpedia, las consultas de AutoComplete se realizarán en el conjunto de datos de DBpedia a partir de entonces, pero para la cadena introducida en el momento de realizar el cambio, ya lanzó la consulta sobre el servidor local. Quizás el usuario, al seleccionar el elemento 'Buscar en DBpedia', pretende que se realice la consulta para la cadena introducida. Por lo tanto, después de efectuar el cambio de fuente de datos, se le da la instrucción de lanzar de nuevo una consulta con el contenido actual de la entrada de texto.

```
var itemSelectEventHandler = function(e)
{
        var aux = e.result.display;
                .
                .
                .

        if(aux.indexOf("New Resource") !== -1)
        {
                        .
                        .
                        .
        }
        else if(aux.indexOf("Look up in DBpedia") !== -1)
        {

                autocomplete.set("requestTemplate", dbpQuery);
                autocomplete.set("source", dbpediaDS);
                autocomplete.sendRequest();
        }
        else
        {
                .
                .
                .
        }
};
autocomplete.on("select", itemSelectEventHandler);
```

Conclusión

En líneas generales, funciona bastante bien y se ha conseguido el objetivo propuesto: modificar la fuente de datos del AutoComplete para realizar la búsqueda en un servidor externo, DBpedia en este caso. No obstante, se han de modificar algunas características.

Ahora el usuario puede buscar datos en el repositorio de DBpedia, pero una vez hecho el cambio no es posible regresar al estado anterior, y es posible que el usuario quiera volver a buscar en el repositorio local.

Por otro lado, la consulta que se lanza a DBpedia depende únicamente del nombre/etiqueta de los recursos. Es necesario incluir una restricción para el tipo de modo que coincida con el rango de la propiedad relacionada para mantener la coherencia entre los datos del repositorio local.

A continuación se incluye una captura de pantalla que muestra como al seleccionar el segundo elemento de la lista de resultados de AutoComplete, se sustituye el conjunto de datos actual por la base de datos de DBpedia, de manera que los recursos ofrecidos como posibles valores para la entrada en edición, generalmente son más numerosos y variados.

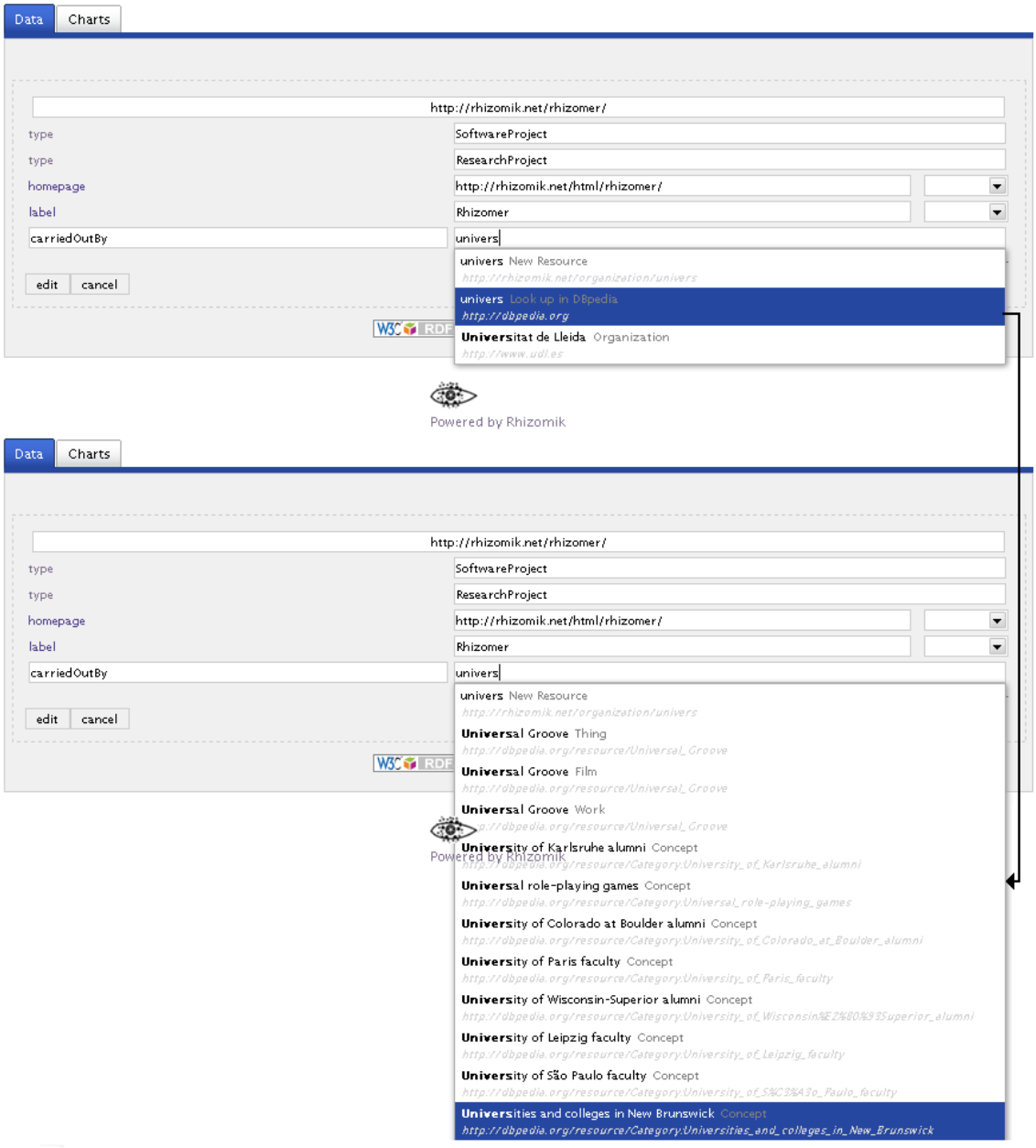

Figura 8: Consultas de AutoComplete en DBpedia

Iteración 14: Bases de datos del AutoComplete intercambiables

Introducción

Se ha añadido la opción de cambiar la fuente de datos del AutoComplete para que busque en DBpedia, pero una vez hecho el cambio no se puede volver a buscar en el servidor local.

Objetivo

Modificar los aspectos necesarios para permitir cambiar de una fuente de datos a otra y viceversa cuando el usuario lo crea oportuno.

Desarrollo

En principio, AutoComplete realiza las consultas sobre el servidor local. El usuario tiene la posibilidad de indicar que prefiere buscar en DBpedia seleccionando el segundo elemento de la lista de resultados proporcionada por AutoComplete. La idea es que una vez haya realizado el cambio y se esté buscando en DBpedia, se pueda volver a realizar las búsquedas en el servidor local de la misma forma, es decir, seleccionando el segundo elemento de la lista de resultados. De esta forma siempre se puede cambiar de un servidor a otro en función de las circunstancias.

Para indicar con qué fuente de datos se está trabajando actualmente, se crea un booleano al que se le modifica el estado cada vez que se procede a cambiar la fuente de datos asignada al AutoComplete. Es decir, si AutoComplete tiene asignada la fuente de datos local, el booleano 'dbpedia' contiene el valor de 'false'; en cambio, cuando AutoComplete está buscando en DBpedia, el booleano 'dbpedia' indicará 'true'. Esto será útil cuando el usuario decida cambiar de base de datos, seleccionará el segundo elemento de la lista de resultados y consultando el booleano sabemos si hemos de asignarle una u otra fuente a AutoComplete.

Primero se configura el segundo elemento de la lista de resultados de AutoComplete para que muestre la información requerida para cada situación. Es decir, si AutoComplete está buscando en el de servidor local, tiene que ofrecer la opción de buscar en DBpedia; y si por el contrario está buscando en DBpedia, el mismo elemento tiene que ofrecer la opción de buscar en el servidor local. De momento sólo se modifica el elemento mostrado al usuario. En el siguiente paso se modificará el comportamiento de AutoComplete al seleccionarlo.

```
var dbpedia = false;
var responseHandler = function(e){

                /* DEFINICIÓN newResource */
                        .
                        .
                        .

        var dbpediaLookUp = {};
        dbpediaLookUp.text = ac_input.get('value');
        dbpediaLookUp.raw = {"label" : {"type" : "literal",
                                        "value" : ac_input.get('value')}};
        if (!dbpedia)
        {
                dbpediaLookUp.display = "<div class='ac-label'>" +
                                        ac_input.get('value') +
                                        "</div><div class='ac-range'>" +
                                        "Look up in DBpedia" +
                                        "</div><div class='ac-uri'>" +
                                        "http://dbpedia.org" +
                                        "</div>";
```

```
        }
        else
        {
                dbpediaLookUp.display = "<div class='ac-label'>" +
                                        ac_input.get('value') +
                                        "</div><div class='ac-range'>" +
                                        "Look up in Local Server" +
                                        "</div><div class='ac-uri'>" +
                                        "http://rhizomik.net" +
                                        "</div>";
        }
        e.results.unshift(dbpediaLookUp);
        e.results.unshift(newResource);
};
autocomplete.on("results", responseHandler);
```

Para modificar el funcionamiento de AutoComplete al seleccionar el segundo elemento, se seguirá el mismo procedimiento que en otras ocasiones. Mediante el evento *select*, se detendrá la ejecución para comprobar si el elemento seleccionado es el segundo. En tal caso, se comprueba qué fuente de datos está asignada a AutoComplete en ese instante, y se procede a modificarle los atributos de necesarios para cambiar de servidor. También se modifica el valor del booleano definido para indicar qué fuente de datos se está utilizando. Una vez realizado el cambio, se lanza una consulta con el valor actual de la entrada de texto asociada para la nueva fuente de datos.

```
var itemSelectEventHandler = function(e)
{
        var aux = e.result.display;
                .
                .
                .

        if(aux.indexOf("New Resource") !== -1)
        {
                        .
                        .
                        .
        }
        else if(aux.indexOf("Look up in ") !== -1)
        {
                if (!dbpedia)
                {
                        dbpedia = true;
                        autocomplete.set("requestTemplate", dbpQuery);
                        autocomplete.set("source", dbpediaDS);
                        autocomplete.sendRequest();
                }
                else
                {
                        dbpedia = false;
                        autocomplete.set("requestTemplate", localQuery);
                        autocomplete.set("source", resourcesDS);
                        autocomplete.sendRequest();
```

```
                }
        }
        else
        {
                .
                .
                .
        }
};
autocomplete.on("select", itemSelectEventHandler);
```

En la siguiente captura de pantalla se muestra como se puede cambiar la fuente de datos asignada al AutoComplete del servidor local a DBpedia y de DBpedia al servidor local indistintamente, seleccionando el segundo resultado de la lista.

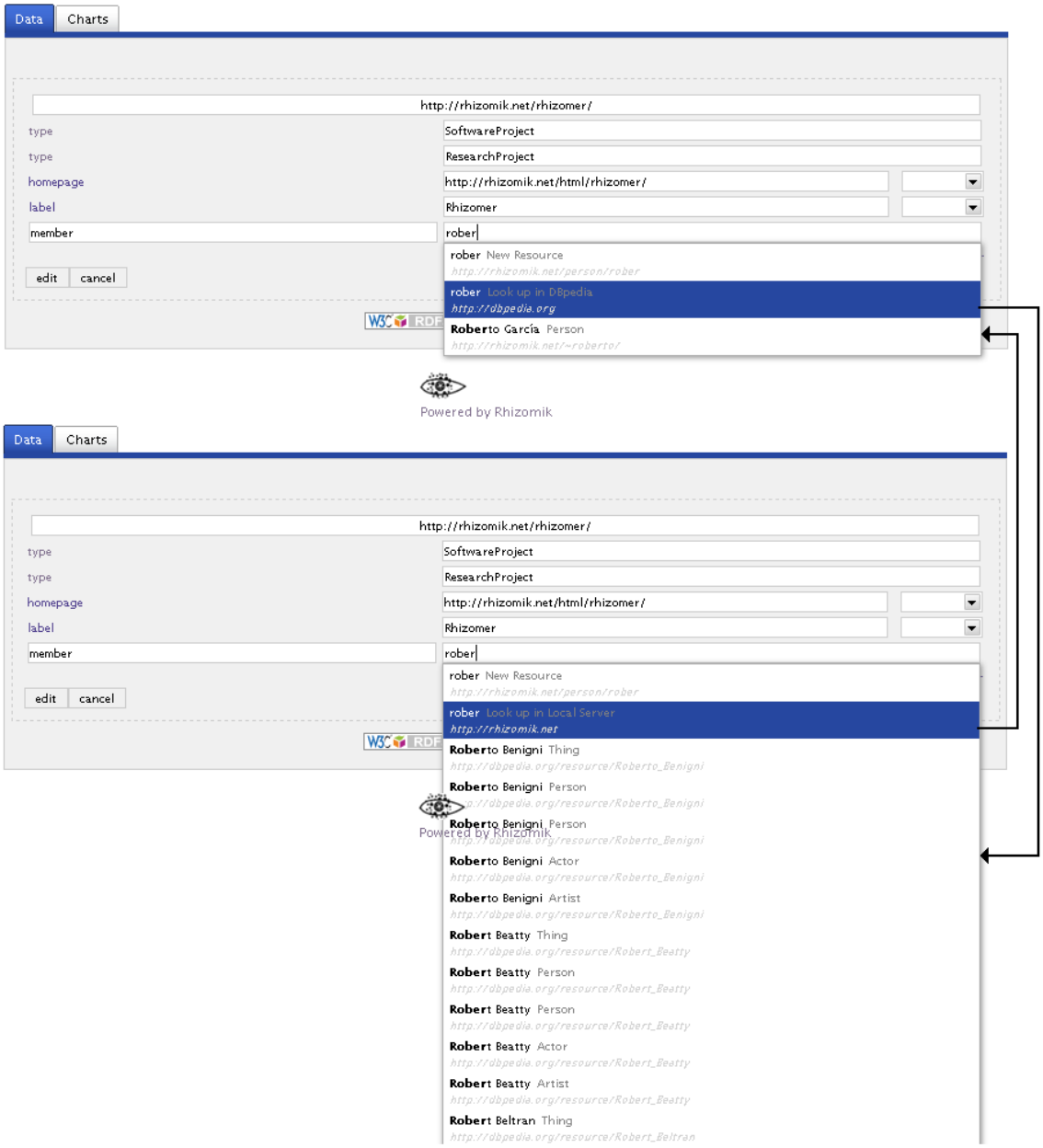

Figura 9: Bases de datos del AutoComplete intercambiables

Conclusión

El sistema implementado para cambiar de base de datos funciona correctamente. Se puede cambiar del servidor local a DBpedia, y de DBpedia al servidor local cuando se crea oportuno, tantas veces como sea necesario.

Iteración 15: Consultas a DBpedia con restricción de tipo

Introducción

Cuando AutoComplete lanza una consulta a DBpedia, ésta contiene restricciones para el atributo 'Label' de los recursos únicamente. Es necesario incluir restricciones de tipo para mantener la consistencia del conjunto de datos, dado que los recursos seleccionados serán posibles valores para la propiedad de un recurso, y cada propiedad tiene un rango definido de manera que solamente puede albergar datos con un dominio específico. Por ejemplo, una supuesta propiedad 'Empleados' puede tener valores de tipo persona o cualquier subclase de ésta, pero sería incoherente que incluyera valores de tipo 'Planta' o 'Lugar', por ejemplo.

Al incluir la restricción de tipo, se observa que en la mayoría de casos es difícil encontrar recursos con exactamente el mismo tipo, ya que éste está definido por una URI, y podemos encontrar la misma clase u ontología definida en distintos sitios. Esto significa que existen múltiples definiciones para referirse al mismo tipo, y cada definición posee un identificador distinto. Por ejemplo, para el tipo 'Person' podemos encontrar fácilmente varias ontologías: http://swrc.ontoware.org/ontology#Person , http://xmlns.com/foaf/0.1/Person, http://dbpedia.org/ontology/Person... y todas representan lo mismo. Por este motivo, se decide añadir una segunda consulta que devuelva todos los recursos que, a parte de la coincidencia del nombre/etiqueta, sin ser del mismo tipo, tienen que contener en la cadena que define al tipo la etiqueta de tipo (la última parte del path de la URI, en este caso 'Person').

Aún con todo, las restricciones son demasiado estrictas, ya que hay muchos recursos que pueden ser valores válidos para la propiedad correspondiente pero no serán seleccionados a través de la consulta indicada. La razón es que hay diferentes formas de llamar a una misma cosa, y por lo tanto, existen ontologías que utilizan nombres distintos para referirse al mismo concepto. Por ejemplo, 'Organization'-' Organisation', 'Football'-'Soccer'… Otro caso que no recoge la consulta SPARQL son las subclases de tipo. Es decir, cuando un tipo de recursos es válido para una entrada, cualquier subclase suya lo es. Si tenemos una propiedad con rango de persona, todos los recursos con tipo de persona son válidos como valor de la propiedad, pero también serán válidos recursos con tipo de arquitecto, actor o profesor, ya que todos ellos también son personas.

Objetivo

Debido a las imprecisiones mencionadas, se decide configurar el AutoComplete para que cuando lance una consulta sobre DBpedia, lo haga dependiendo de los resultados obtenidos en la última consulta.

Primeramente lanzará la consulta SPARQL con restricción de tipo. Si no se obtienen resultados, el patrón de consulta de AutoComplete, almacenado en su atributo

requestTemplate, se modifica para cambiar la consulta de manera que en lugar de restringir el tipo, se establece que éste contenga la etiqueta que se refiere a él (e.g. 'Person').

Si con el segundo patrón de consulta tampoco se obtienen resultados, se vuelve a modificar el atributo correspondiente de AutoComplete para realizar, esta vez, una consulta sin restricción de tipo. Solamente se comprueba que el inicio del campo 'Label' coincida con la cadena introducida, de forma que se obtendrán resultados de múltiples tipos, y no todos serán apropiados para el campo en edición. De todas formas, se muestra claramente el tipo de cada recurso en la lista de resultados, de modo que el usuario es quien tiene que decidir el recurso que quiere especificar como valor para la propiedad relacionada.

Desarrollo

Primero se tienen que definir las funciones encargadas de generar la consulta SPARQL codificada. Se utiliza una sola función para definir los tres patrones, el que restringe la URI del tipo, el que restringe la etiqueta del tipo y el que no tiene restricciones de tipo. Para ello se definen tres variables que representan cada una una condición para la consulta SPARQL. De esta forma se aprovecha la función para generar tres consultas distintas a partir de la misma consulta, ya que contiene una condición intercambiable que se modificará según las necesidades del sistema.

Se puede observar como, en la cadena que se devuelve con la consulta codificada, además de la consulta SPARQL, se especifica un parámetro para el formato requerido de respuesta y otro para el tiempo límite.

```
var queryTypeUriString = "(?type = <" + resourceType +  ">) &&";
var queryTypeLabelString = "(regex(?type, ' " + rlabel + " ', 'i')) &&";
var genericQuery = "";
var resourceType2 = queryTypeUriString;

function dbpQuery(sQuery)
{
        var queryPattern =    " SELECT DISTINCT ?uri ?label ?type
                              WHERE {
                              ?uri rdf:type ?type; rdfs:label ?label.
                              FILTER ( + resourceType2 +
                              (REGEX(?label, '( |^) + sQuery + .*', 'i'))))}
                              LIMIT 500 ";
        return '&query=' + encodeURIComponent(queryPattern) +
                                                '&format=json&timeout=5000';
}
```

Una vez creada la consulta con la condición intercambiable, se procede a especificar cuando se debe lanzar una u otra consulta. En resumen, cuando se cambia la fuente de datos del servidor local a DBpedia, funciona con la consulta de restricción de tipo (URI del tipo). Si no se obtienen resultados para la consulta, se cambia el patrón para realizar consultas restringiendo la etiqueta del tipo. Y si de nuevo no hay resultados, se cambia a la consulta genérica sin restricción de ninguna clase para el tipo.

Por lo tanto, primero de todo se comprueba si la fuente de datos actual que utiliza AutoComplete es DBpedia en la función *responseHandler()*, que se ejecuta cuando se produzca el evento *results*, es decir, cuando se recibe un conjunto de resultados. Luego se comprueba si el conjunto de resultados recibidos está vacío. En tal caso se ha de cambiar la consulta, pero se han de cumplir dos condiciones más para que sea necesario el cambio de patrón:

- El patrón actual debe ser distinto al patrón de consulta sin restricción de tipo, ya que en tal caso se trata de la consulta más genérica y es la última opción.

- El atributo 'data' del evento no debe ser *undefined*. Este atributo contiene los datos recibidos, es decir, el conjunto de resultados. Cuando no hay resultados es un conjunto de datos vacío, pero definido. En cambio, cuando Virtuoso lanza una excepción, este atributo no está definido. Y que lance una excepción no implica que no haya resultados, y si hay resultados no se tiene que cambiar todavía el patrón de consulta por uno menos específico.
  Las excepciones más comunes son debido a que se requieren más de tres caracteres para lanzar la consulta, o que no se pueden obtener más de mil resultados. La primera se soluciona fácilmente restringiendo las consultas de AutoComplete a un mínimo de 4 caracteres. En cambio, para resolver la segunda, se ha especificado en la consulta SPARQL que limite la búsqueda a 500 resultados, pero aún así, sigue sucediendo.

Una vez que se ha confirmado que se dan todas las condiciones necesarias para el cambio de patrón de consulta, se comprueba la condición para el patrón de consulta actual y se le asigna la siguiente condición. Es decir, si se está usando la condición para la restricción de la URI del tipo, se cambia por la condición que restringe la etiqueta del tipo. Y si se está empleando la condición para la etiqueta del tipo, se asigna la condición genérica.

```
var responseHandler = function(e)
{
                .
                .
                .

        if (!dbpedia)
        {
                .
                .
                .
        }
        else
        {
                dbpediaLookUp.display = "<div class='ac-label'>" +
                                        ac_input.get('value') +
                                        "</div><div class='ac-range'>" +
                                        "Look up in Local Server" +
                                        "</div><div class='ac-uri'>" +
                                        "http://rhizomik.net" +
                                        "</div>";
                if(e.results.length === 0 && resourceType2 !== genericQuery && e.data
                                                                        !== undefined)
                {
```

```
                        if (resourceType2 === queryTypeUriString)
                        {
                                resourceType2 = queryTypeLabelString;
                        }
                        else if (resourceType2 === queryTypeLabelString)
                        {
                                resourceType2 = genericQuery;
                        }
                        autocomplete.sendRequest();
                }
        }
        e.results.unshift(dbpediaLookUp);
        e.results.unshift(newResource);
};
autocomplete.on("results", responseHandler);
```

Por último, falta modificar la función que da formato a la lista de resultados para que muestre correctamente el tipo de cada dato.

Cuando AutoComplete trabaja con el servidor local, la etiqueta del tipo, que es lo que queremos mostrar, se encuentra definida en la variable 'rlabel', que contiene la etiqueta del rango de la propiedad asociada a la entrada que se está editando. En este caso, la consulta SPARQL siempre incluye la restricción de tipo, por lo que se tiene la garantía de que los datos recibidos son del tipo indicado.

Cuando AutoComplete trabaja con DBpedia, es necesario consultar el tipo directamente de los datos recibidos. Como se pretende mostrar la etiqueta del tipo y éste generalmente es una URI, se extrae la última sección del path de la URI del tipo.

```
autocomplete.set("resultFormatter", function(query, resultsArray)
{
        return Y.Array.map(resultsArray, function(result)
        {
                var resultTemplate = "<div class='ac-label'>" +
                                     "{label}" +
                                     "</div><div class='ac-range'>" +
                                     "{type}" +
                                     "</div><div class='ac-uri'>" +
                                     "{uri}" +
                                     "</div>";
                var resultJSON = result.raw;
                if (dbpedia)
                {
                        if(rhizomik.Utils.isURI(resultJSON.type.value))
                        {
                                resourceType3 =
                                        rhizomik.Utils.uriLocalname(resultJSON.type.value);
                        }
                        else
                        {
                                resourceType3 = resultJSON.type.value;
                        }
                else
                {
```

```
                        resourceType3 = rlabel;
                }
                return Y.Lang.sub(resultTemplate, {
                                label           : result.highlighted,
                                type            : resourceType3,
                                uri             : resultJSON.uri.value
                                });
        });
});
```

Una vez se ha comprobado que el sistema para el cambio de consulta funciona correctamente, se decide modificar su funcionamiento para que cuando el usuario borre algún carácter de la entrada, se reinicie el patrón de consulta para restringir el tipo.

Hasta ahora, una vez que se cambiaba a una consulta menos específica, no se regresaba al estado anterior ya que lo más probable es que el usuario continúe escribiendo sobre la misma cadena, por lo que no se obtendrían resultados con la consulta anterior. Pero cabe la posibilidad que el usuario decida cambiar la palabra introducida, y en tal caso se debería restablecer el patrón de consulta para realizar la búsqueda con restricción de tipo.

Por lo tanto, se decide añadir una escucha sobre la entrada asociada para que cuando el usuario borre algún carácter (señal de que quiere modificar el contenido), se realicen los cambios necesarios para comenzar la búsqueda desde cero, es decir, con restricción de tipo.

La escucha se implementa a través de un evento de YUI 3, que detecta la presión sobre cualquier tecla del teclado. Las teclas que se pretenden escuchar para detectar la supresión de algún carácter son el 'retroceso' (backspace) y el 'suprimir' (del/delete), y se especifican en el tercer argumento de la suscripción mediante sus 'keycodes'.

```
        var bs_delHandler = function(e)
        {
                if (dbpedia && resourceType2 !== queryTypeUriString)
                {
                        resourceType2 = queryTypeUriString;
                }
        };
        ac_input.on("key", bs_delHandler, 'down: 8, 127');
```

En la siguiente captura de pantalla, se destaca el tipo de los datos recibidos de una consulta a DBpedia para mostrar como las restricciones de tipo incluidas funcionan correctamente.

El rango de la propiedad 'member', cuyo valor es el que se está definiendo, es 'Person' (http://swrc.ontoware.org/ontology#Person). Por lo tanto, si existen recursos en el conjunto de datos de DBpedia con tipo de persona, solamente deben mostrarse éstos.

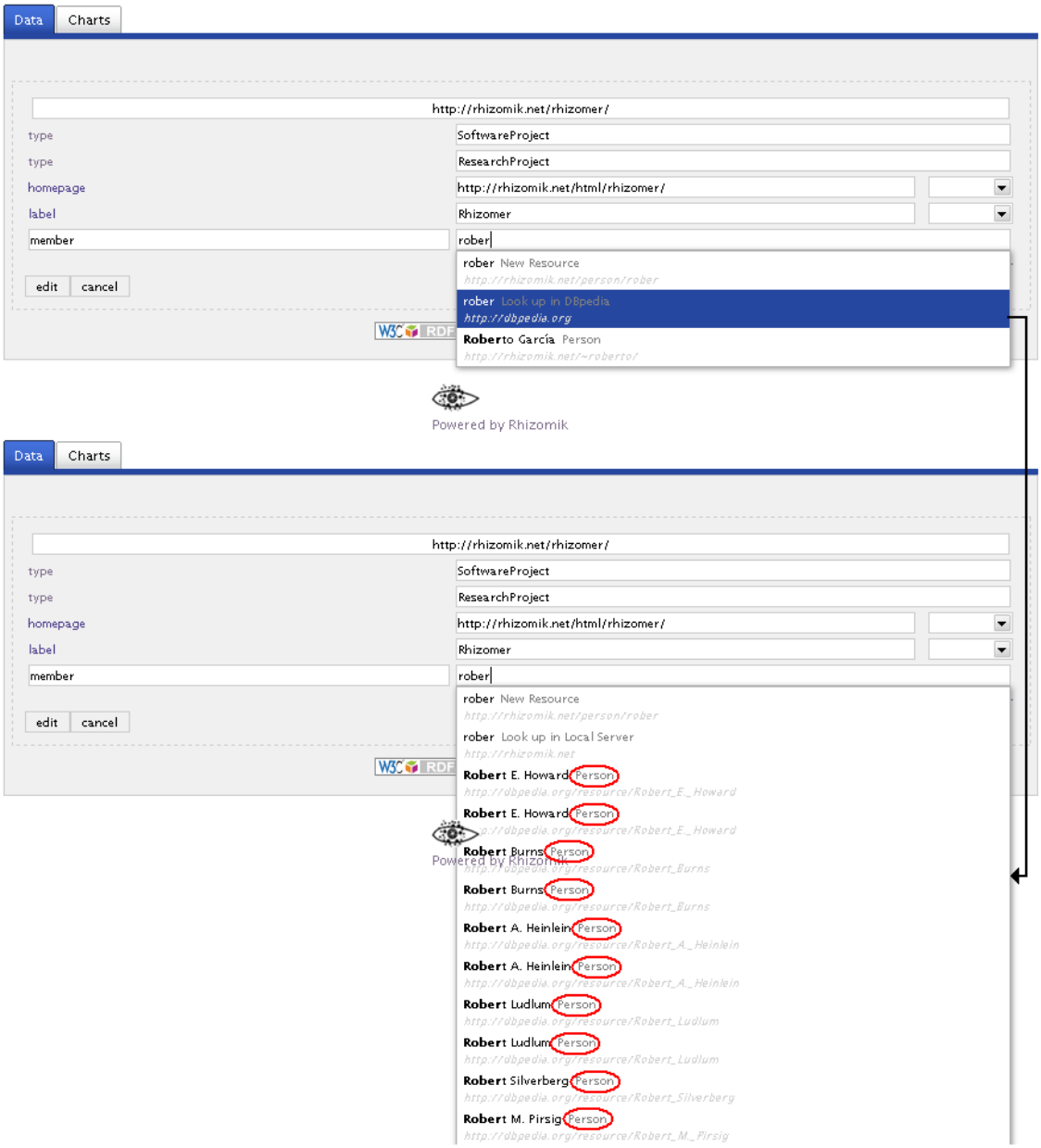

Figura 10: Consultas a DBpedia con restricción de tipo

Conclusión

Aunque han surgido varias dificultades, se puede decir que se ha conseguido implementar el funcionamiento propuesto. AutoComplete puede utilizar como base de datos un servidor externo, DBpedia, y además se han integrado distintos niveles de restricción para las consultas en función de los resultados obtenidos.

Además, es fácilmente configurable para cambiar a otra fuente de datos externa, siempre y cuando funcione con Virtuoso, simplemente hace falta cambiar una URL.

No obstante, hay que decir que va más lento de lo esperado, el tiempo de respuesta se considera excesivo. Por otro lado, Virtuoso lanza demasiadas excepciones, cosa que complica el funcionamiento de la herramienta pudiendo provocar confusión en el usuario. Se barajan alternativas para solucionar ambos casos y se expondrán en la sección de trabajo futuro.

Iteración 16: Refactoring

Introducción

Durante el desarrollo del proyecto, se han modificado y añadidos varias funcionalidades en la herramienta Rhizomer, concretamente en los formularios de edición. Para ello se ha ampliado la clase 'SemanticForms' y se han modificado algunos aspectos, y generalmente se ha trabajado sobre tres métodos contenidos en la clase mencionada. Como consecuencia, ha llegado un punto en que dichos métodos son demasiado extensos y complicados, lo que hace difícil la lectura y comprensión del código comprendido.

Objetivo

Organizar y estructurar el código contenido en la clase 'SemanticForms' en distintas clases y métodos, para conseguir código más legible, ordenado y escalable.

Desarrollo

Es difícil detallar el trabajo hecho durante esta iteración, ya que se ha modificado considerablemente toda la clase 'SemanticForms', y para exponer todos los cambios realizados, se tendría que explicar de nuevo todo el trabajo realizado durante la segunda etapa del proyecto, ya que, aunque el resultado final sea el mismo, el código que interviene está desglosado en distintos métodos, cada uno de los cuales debería explicarse por separado.

Dado que el funcionamiento de la aplicación es idéntico, considero innecesario volver a explicar detalladamente el trabajo realizado. En su lugar, se incluye un diagrama de métodos para hacerse una idea del cambio realizado sobre el código expuesto. De todas formas, todo el código generado está disponible en http://code.google.com/p/rhizomer/source/browse/src/main/webapp/script/rhizomer-forms.js y http://code.google.com/p/rhizomer/source/browse/src/main/webapp/script/autocomplete.js

No obstante, cabe destacar que se ha creado una clase nueva, 'AutoComplete', con varios métodos para la creación, configuración y manipulación de entradas con autocompletar. En la clase SemanticForms, se ha desglosado el código en múltiples métodos, todos ellos cortos, sencillos y fácilmente entendibles. Aunque hay algunos métodos generados para acciones puntuales o circunstancias específicas, la mayoría de métodos se han implementado de forma genérica, para que puedan ser utilizados en distintas situaciones.

A continuación se muestra una relación de los tres principales métodos empleados durante el desarrollo del proyecto, cada uno de los cuales contenía infinidad de líneas de código. Se pretenden reflejar los nuevos métodos que cada uno de los "antiguos" llama para efectuar el trabajo que ellos mismo hacían antes, conteniendo todo el código bajo la misma función. Aunque se ha reestructurado la mayoría de código de la clase SemanticForms, se incluyen los tres principales métodos en los se ha trabajado, ya que son los que contienen cambios relevantes.

Los métodos enlazados con línea continua, se ejecutarán siempre que la función que los ha llamado se ejecute. En cambio, los métodos enlazados con líneas punteadas se ejecutarán en función del estado de algunos parámetros del formulario o de alguno de sus elementos (entradas, autocomplete's…)

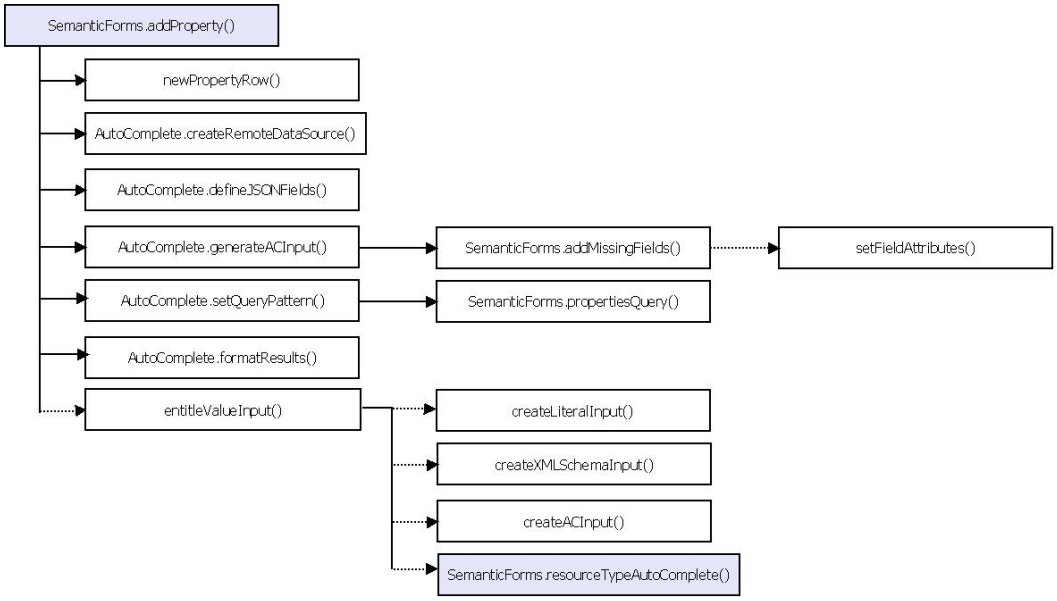

Figura 11: Refactoring - addProperty()

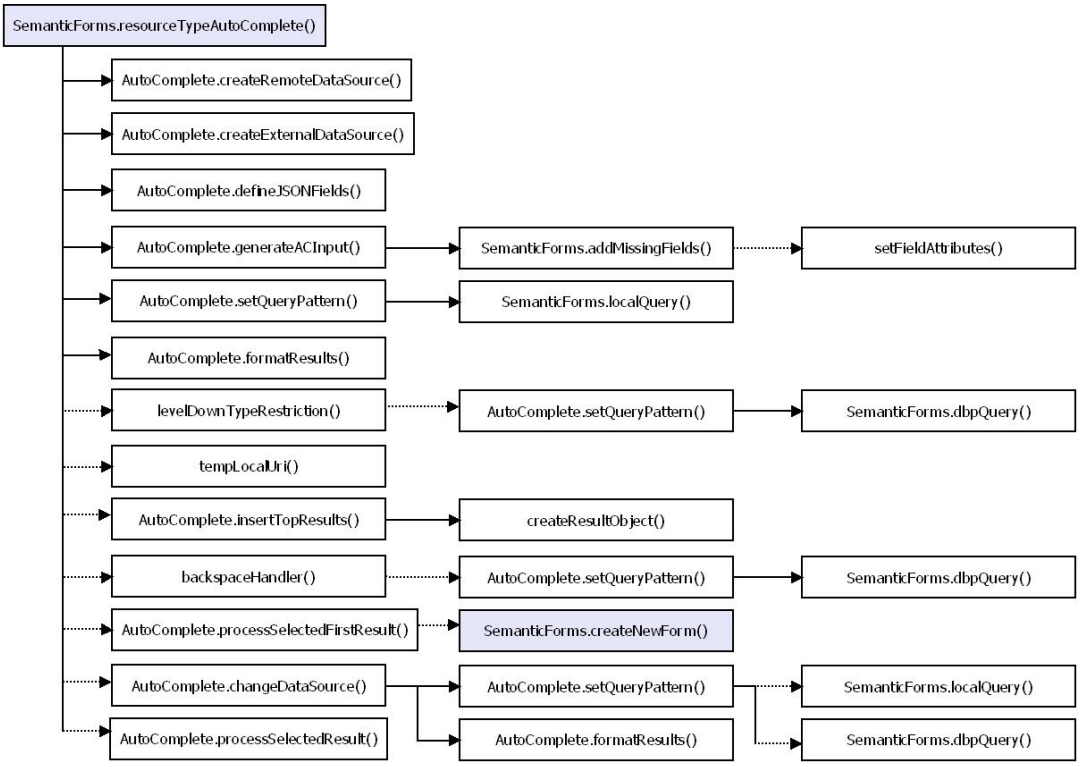

Figura 12: Refactoring – resourceTypeAutoComplete()

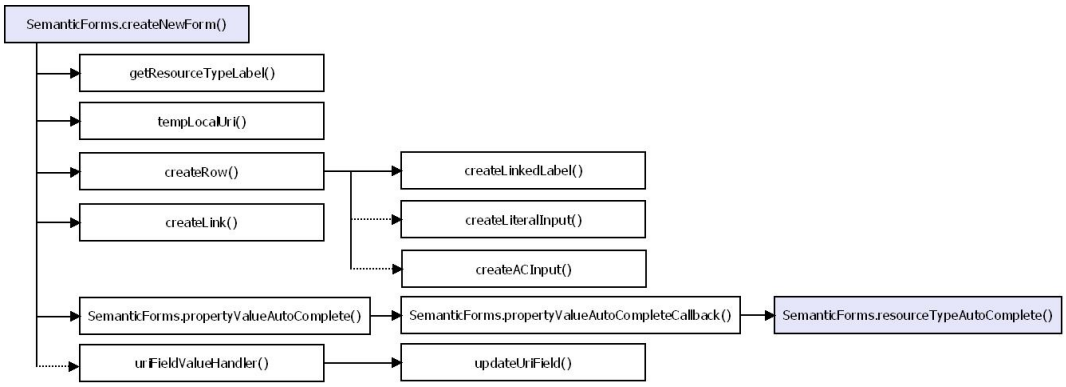

Figura 13: Refactoring - createNewForm()

Conclusión

Hasta la fecha, se habían ido añadiendo funcionalidades y demás líneas de código sobre los tres mismos métodos, llegando a un punto en que el código resultaba difícil de comprender. Se ha reestructurado el código de forma que todos los métodos obtenidos sean simples y entendibles, varios de los cuales se utilizan en distintas ocasiones y en distintas circunstancias, ya que se han intentado generar de forma que sean lo más genéricos y escalables posible. Por lo tanto, se puede decir que el cambio realizado ha sido tan necesario como provechoso.

Bloque de finalización

Conclusiones

Por un lado, comentar que el trabajo realizado durante la primera etapa del proyecto, es decir, el formulario dinámico de consulta, ha sido sustituido por otros sistemas de localización de datos. De todas formas, el trabajo realizado ha sido de utilidad para la aplicación, dado que los formularios de edición actuales disponen de las principales características del formulario dinámico de consulta adaptadas a las nuevas circunstancias (entradas con autocompletar y posibilidad de personalizar el formulario agregando entradas adicionales).

Durante el desarrollo de la segunda etapa del proyecto se han ido definiendo distintos objetivos en función de las necesidades o carencias de la aplicación sobre la que se trabaja. Se puede decir que se han cumplido satisfactoriamente todas las tareas propuestas: creación de un formulario para la definición de descripciones semánticas, actualización del código incluido de las librerías de Yahoo! User Interface Library a la última versión, modificación del objeto AutoComplete para que trabaje con una fuente de datos remota, posibilitar la búsqueda de recursos para la definición de descripciones semánticas en conjuntos de datos localizados en dominios externos a la aplicación…

En definitiva, el propósito general del proyecto era conseguir una publicación de datos semánticos más sostenibles para aprovechar la intervención del usuario con el fin de mejorar la calidad de los datos. Esto se ha conseguido gracias al desarrollo de mecanismos de edición con asistencia al usuario. Durante el proceso de edición de un recurso, los mecanismos de asistencia ofrecen al usuario varias alternativas para especificar el valor de sus propiedades: recursos existentes en el repositorio local de datos, recursos obtenidos de conjuntos de datos de referencia (i.e., DBpedia), o bien crear un nuevo recurso mediante la especificación de su descripción semántica a través de formularios generados dinámicamente. Los formularios generados contienen un conjunto de propiedades definidas, pero editables. El mecanismo de asistencia para definir la descripicón semántica del nuevo recurso es el mismo que en el caso anterior, por lo tanto, se dispone nuevamente de las tres vías citadas para editar cada valor.

No obstante, el sistema para obtener recursos del conjunto de datos de DBpedia no funciona del todo como se esperaba. Aunque su funcionamiento es correcto, y no produce errores, las consultas sobre la base de datos se hacen a través de Virtuoso, y éste, en algunos casos, lanza excepciones que alteran ligeramente el funcionamiento de la aplicación, lo que puede desorientar al usuario, provocando confusión y como consecuencia, se reduce la usabilidad de la herramienta. Además, va más lento de lo esperado, cuando se solicita un conjunto de datos de un servidor externo, el tiempo de respuesta se considera excesivo.

A nivel personal, la elaboración y el desarrollo del proyecto me ha aportado mucho más de lo que esperaba. En primer lugar, he aprendido a utilizar lenguajes y tecnologías con gran salida laboral, como JavaScript, AJAX, o la Web Semántica, lo que me puede facilitar la incorporación laboral al mundo de la informática. Pero sobre todo, bajo mi punto de vista, deducido de todo el material estudiado a lo largo del proyecto, las tecnologías que he aprendido van a tener una gran presencia en el futuro de la Web y de la informática.

En segundo lugar, a parte de aprender algunos lenguajes de programación, el desarrollo de este proyecto ha sido especialmente útil para acumular, a base de práctica, la confianza y soltura necesarias para programar con cierta autonomía, buscar motivos y soluciones a los problemas que se presenten… En definitiva, he aprendido a desenvolverme por entornos Web y aplicaciones, en mayor o menor medida, pero sin duda alguna, notablemente mejor que con los conocimientos adquiridos y las prácticas realizadas a lo largo de la carrera.

Trabajo futuro

En esta sección se exponen algunas extensiones que quedan pendientes de implementar en un futuro para mejorar el resultado obtenido o para ampliar las funcionalidades y servicios ofrecidos. También se incluye algún aspecto implementado durante el proyecto susceptible de mejorar, de los que no se ha obtenido el resultado esperado, o bien posteriormente se ha decidido realizar alguna modificación.

- Recompilar una copia local de Virtuoso para modificar la restricción de 1000 resultados máximo, de modo que en lugar de lanzar una excepción y anular la consulta, se tengan en cuenta los 1000 primeros resultados. De esta forma, cuando haya más de mil resultados, se recibirán los mil primeros en lugar de un conjunto vacío de datos, lo que puede llevar al usuario a la confusión porque se puede interpretar que no hay resultados para la consulta realizada, cuando en realidad hay más de mil.

- Recompilar la copia local de Virtuoso para modificar la instrucción 'bif:contains' de modo que tome toda la consulta como una cadena. Esta instrucción compara palabra por palabra, y Virtuoso restringe la consulta mínima a cuatro caracteres. Esto implica que todas las palabras de la consulta deben superar los tres caracteres, en cualquier otro caso no se envía la petición. Esto comporta un funcionamiento ilógico, ya que, por ejemplo, para la palabra 'Nueva', probablemente se obtendría un conjunto de resultados, pero al continuar escribiendo → 'Nueva Y', la nueva consulta no cumple las restricciones impuestas, por lo que esta vez no se recibirán resultados dado que la consulta no se lanza.

  La razón para tal modificación es que la instrucción 'bif:contains' funciona notablemente más rápido que 'regex' (es el filtro para restringir cadenas que se usa en el código actual), de hecho, en un principio es la que se utilizaba, y el tiempo de respuesta era aceptable. Pero dadas las incoherencias mencionadas que implica su uso, tuvo que ser sustituida, y aunque de esta forma se han solventado los problemas comentados, el tiempo de respuesta obtenido es demasiado grande.

- El formulario para definir descripciones semánticas contiene una entrada 'Type' con un valor asignado, definido a partir del rango de la propiedad que se está editando. No obstante, dicho valor es editable, y la entrada en cuestión funciona con autocompletar, mostrando todos los tipos definidos en las ontologías utilizadas que coincidan en su inicio con la cadena introducida. Se pretende modificar la función de auto-completado de esta entrada en concreto, para que muestra solamente las subclases del tipo predefinido, ya que el recurso asignado como valor de una propiedad, tiene que ser de un tipo específico, definido por el rango de la propiedad, o bien de alguna subclase del tipo indicado, ya que éstas también pertenecen al tipo.

- Modificar el comportamiento del enlace para añadir propiedades del formulario de definición de descripciones semánticas cuando el valor de la entrada 'type' sea modificado, para que las propiedades se obtengan en función del nuevo tipo definido.

- Modificar el formulario para la definición de descripciones semánticas para que además de las propiedades URI, Label y Type, contenga en el momento de su generación, un conjunto de propiedades obtenidas a partir de comparar otros recursos de la misma clase y comprobar qué propiedades asignadas presentan de forma más frecuente.

- Modificar el sistema para añadir propiedades de forma que al seleccionar una de las propiedades mostradas a través de AutoComplete, la entrada de texto implicada sea sustituida por una etiqueta enlazada a la descripción semántica de la propiedad seleccionada.

- Implementar un formulario especializado para la edición de algunas estructuras de RDF, como por ejemplo listas, owl:Restrictions…

Bibliografía

- Libros
  - Pollock, Jeffrey T. (2009). *Semantic Web for Dummies*. Wiley Publishing, Inc.
  - Crockford, Douglas (2008). *JavaScript, The Good Parts*. O'Reilly.
  - Stefanov, Stoyan (2010). *JavaScript Patterns*. O'Reilly.
  - Eguíluz Pérez, Javier (2009). *Introducción a JavaScript*. www.librosweb.es.
  - Holzner, Steve (2006). *AJAX for Dummies*. Wiley Publishing, Inc.
- Artículos
  - Brunetti, Josep Maria; Gil, Rosa; López-Muzás, Antonio; Gimeno, Juan Manuel; García, Roberto (2011). *Evaluación de una plataforma semántica para la Interacción con la Web de Datos*. XII Congreso Internacional de Interacción Persona Ordenador (Interacción 2011), Lisboa, Portugal.
  - Castells, Pablo (2003). *La web semántica*. Escuela Politécnica Superior, Universidad Autónoma de Madrid.
- Material Docente
  - del Teso, Enrique (2007). *Tecnología XML y Web Semántica. Ontologías*. Dto. Filología Española, Universidad de Oviedo.
- Referencias Web
  - González Ortega, Fco. Javier (2007). *El lenguaje RDF*. Disponible en: http://serqlsparql.50webs.com/rdf.html
  - W3C (2008a). *SPARQL Quey Language for RDF. W3C Recomendation 15 January 2008.* Disponible en: http://www.w3.org/TR/rdf-sparql-query/
  - W3C (2008b). *SPARQL Quey Results XML Format. W3C Recomendation 15 January 2008.* Disponible en: http://www.w3.org/TR/rdf-sparql-XMLres/
  - W3C (2008c). *SPARQL Protocol for RDF. W3C Recomendation 15 January 2008.* Disponible en: http://www.w3.org/TR/rdf-sparql-protocol/
  - Garret, Jesse James (2005). *Ajax: A New Approach to Web Applications*. Disponible en: http://www.adaptivepath.com/ideas/ajax-new-approach-web-applications

- van Kesteren, Anne (2012). *XMLHttpRequest. W3C Working Draft 17 January 2012*. Disponible en: http://www.w3.org/TR/XMLHttpRequest/

- Zaera Avellón, Iván (2006). *Introducción a la tecnología AJAX*. Disponible en: http://www.adictosaltrabajo.com/tutoriales/tutoriales.php?pagina=beginAjax

- Yahoo! Inc. (2012). *Yahoo! User Interface Library*. Disponible en: http://yuilibrary.com

- W3C (1999). *HTML 4.01 Specification. W3C Recommendation 24 December 1999*. Disponible en: http://www.w3.org/TR/REC-html40/